



Proceedings of VikingPloP 2012 Conference

Citation

Eloranta, V-P., Koskinen, J., & Leppänen, M. (Eds.) (2012). Proceedings of VikingPloP 2012 Conference. (Tampere University of Technology. Department of Software Systems. Report; Vol. 22). Tampere University of Technology. Department of Software Systems.

Year

2012

Version

Publisher's PDF (version of record)

Link to publication

TUTCRIS Portal (<http://www.tut.fi/tutcris>)

Take down policy

If you believe that this document breaches copyright, please contact cris.tau@tuni.fi, and we will remove access to the work immediately and investigate your claim.

Veli-Pekka Eloranta, Johannes Koskinen & Marko Leppänen (eds.)
Proceedings of VikingPLoP 2012 Conference



Veli-Pekka Eloranta, Johannes Koskinen & Marko Leppänen (eds.)

Proceedings of VikingPLoP 2012 Conference

ISBN 978-952-15-2943-6 (printed)
ISBN 978-952-15-2944-3 (PDF)
ISSN 1797-836X

Preface

You are now reading the proceedings of VikingPLoP 2012. VikingPLoP is a Nordic conference of pattern languages of programs which last time took place in Saariselkä, Lapland, Finland in March 2012. VikingPLoP was organized by Tampere University of Technology, Hillside Europe, and Cones after a few years of hiatus. Saariselkä was chosen as a venue location as it is in the middle of ruggedly beautiful wilderness of Lapland. During the conference, there was still plenty of snow, but the sun was also warming up the days.

The papers in this proceedings are updated versions of the papers workshopped in the conference. Participants submitted their papers for shepherding process. In shepherding process, experienced pattern writer gave ideas and feedback for the author, colloquially known as a sheep. The sheep incorporated this feedback in to her paper. After three iterations of shepherding the paper was discussed at the conference in writer's workshop. Workshop group gave comments, criticism and praise. After the conference sheep updated their papers according to the workshop feedback. This process of giving feedback was made possible by having community of trust. Mutual trust was built by playing non-competitive games and by having social activities.

VikingPLoP 2012 focused on patterns and their usage in various fields of expertise. These fields included a wide range of topics from language teaching to embedded system's software architecture. Bringing people together from various fields of expertise, stimulates creativity and new ideas might emerge. These innovations are reflected in the papers in these proceedings. VikingPLoP 2012 was especially a conference for newcomers and over half of the participants were first time PLoP participants.

These proceedings contain 10 papers and description of one focus group. In addition, a shepherding workshop was arranged and updated version of the demo pattern used in this workshop is also presented in the proceedings. The conference had two writer's workshop groups. Papers are organized as follows: in the first part of the proceedings patterns for embedded systems are presented and the second part contains general software related patterns. Finally in the third part, interdisciplinary patterns are included.

As expected, VikingPLoP 2012 was an enjoyable and fun experience. If you contributed to it in any way, we are grateful for your involvement. If you wish to participate in the future, please come and find out more information about the next conference <http://www.vikingplop.org> and join the community. We wish that these proceedings are a valuable source of information in your efforts. We hope that you will enjoy reading the following pages.

October 2012

The program chairs,

Veli-Pekka Eloranta and Marko Leppänen

Thanks

We would like to send out our thanks to everybody who has helped us to organize this event:

- the authors, who have written and submitted their papers to the conference;
- the shepherds, whose feedback to the submitted papers have helped the authors to be accepted to the conference;
- the program committee, whose help has been invaluable;
- our sponsors, who have financially supported our efforts;
- and the staff of Tampere University of Technology who have provided the cabin and have helped us in all the small organizational tasks.

Organization

VikingPLoP is a non-profit event organized by Tampere University of Technology with support from Hillside Europe and various individuals from pattern community.

Program chairs

- Veli-Pekka Eloranta, Tampere University of Technology, Finland
- Marko Leppänen, Tampere University of Technology, Finland

Program committee

- Ville Reijonen, Kauppalehti Online Development, Finland
- Johannes Koskinen, Tampere University of Technology, Finland
- Juha Pärssinen, VTT, Technical Research Centre of Finland, Finland
- Kai Koskimies, Tampere University of Technology, Finland
- James O. Coplien, Gertrud & Cope, Denmark

VikingPLoP supporters

- NordForsk: [CONES - Center of Nordic Excellence in Software Engineering](#). VikingPLoP 2012 was also CONES event.
- Klaus Marquardt
- Dietmar Schutz
- Kristian Elof Sørensen
- Joe Yoder
- Neil Harrison
- Tim Wellhausen
- Tampere University of Technology - Department of Software Systems
- Hannu-Matti Järvinen

Shepherds

Neil Harrison, Juha Pärssinen, Farha Lakhani, Uwe van Heesch, Veli-Pekka Eloranta, Marko Leppänen, Johannes Koskinen, James O. Coplien

Table of Contents

<i>Sandra Lindberg</i> Communicating the feel of design	1
<i>Veli-Pekka Eloranta</i> Demo workshop: Bulletin board	2

PART I – EMBEDDED SYSTEMS

<i>Veli-Pekka Eloranta</i> Event Notification Patterns for Distributed Machine Control Systems	4
<i>Johannes Koskinen</i> Patterns for High Availability Distributed Control Systems	23
<i>Marko Leppänen</i> Patterns for Distributed Machine Control System Fault Tolerance Modes	36
<i>Jari Rauhamäki, Timo Vepsäläinen, and Seppo Kuikka</i> Functional Safety System Patterns	48
<i>Ville Reijonen</i> Patterns Related to Software Updating for Machine Control Systems	69

PART II – SOFTWARE ENGINEERING

<i>Ilkka Laukkanen</i> Doing Small-scale Agile Projects Efficiently and Profitably	80
<i>Pietu Pohjalainen</i> Self-configurator	86
<i>Joonas Salo</i> Pattern definition of MapReduce	97

PART III – DIALOGUE PATTERNS

<i>Christian Köppe and Mariëlle Nijsten</i> Towards a Pattern Language for Teaching in a Foreign Language	109
<i>Dirk Schnelle-Walka, Stefan Radomski</i> A Pattern Language for Dialogue Management	122

Communicating the feel of design

Sandra Lindberg

Institutiaon of communication, media and ICT, Södertörns university, Huddinge, Sweden
sandra.k.lindberg@gmail.com

In my masters thesis work in interactive media design at Södertörn university Sweden, I was exploring how communication between interaction designers and developers can be supported by using design patterns. Through a pre-study survey I aimed to investigate how the communication between designers and developers takes place today and what role design patterns play in this communication.

From the results of the survey it was possible to pinpoint that the biggest difficulty in the communication between designers and developers is to convey/understand the feel of a design, and that the most common tool to facilitate the communication is prototypes, sketches and textual descriptions. Rather few participators used the design pattern libraries and the result showed no connections between using design patterns and having an easier communication between developers and designers.

At VikinPLoP I held a focus group where we discussed the lack of technical/constraints knowledge among designers, how to convey/understand the feel of a design and the lack of a common vocabulary. When dealing with lack of technical/constraints knowledge a common vocabulary and design patterns were suggested for facilitating a mutual understanding between designers and developers. However before using design patterns as a tool in the communication the common vocabulary needs to be established. The common vocabulary could consists of glossaries and synonyms of words which occur in both fields, other suggestions were to use scenarios or user stories which can be understood by both designers and developers as a tool for facilitating communication. To facilitate the communication of the feel of a design, it was suggested that illustrations such as moodboards is the best way to convey a feel. The moodboards could be presented in a design pattern. A design pattern for the feel of design could also be complemented with a compare and contrast section, which would show examples of extreme opposites e.g. light as a feather vs. heavy as a stone.

The result of the prestudy and the focus group at VikingPLoP was the ground for my further work with an article which aims to contribute to the unexplored field of affective aesthetics by presenting and evaluating a design proposal for a tool, which aims to facilitate the communication around affective aesthetics in design teams.

Demo workshop: Bulletin board

Veli-Pekka Eloranta

Tampere University of Technology, Department of Software Systems

P.O. Box 553, FI-33101 Tampere, Finland, veli-pekka.eloranta@tut.fi

1 Bulletin board pattern

... you have an organization where many people work. There are all kinds of events happening in the organization, for example training, lectures, group meetings, team sports etc. Furthermore, there might be other kind of information that needs to be shared to other people in the organization, e.g. announcements. Information being shared may not be so important for everyone that all the people should be notified by e-mail or phone. The group where the information needs to be shared is located in one floor and the number of people is relatively small, e.g. under 100 people. Additionally, there is a central place where people gather once in a while, for example for coffee breaks.

How to efficiently share information between organization members?

Efficient information sharing is a key factor in building successful organization.

People are disrupted from their work when they receive phone calls and e-mails. Therefore these methods should be avoided, whenever possible.

Any member of an organization should be able to share information. In this way people feel they are equal and appreciated.

The information should be easily available for everyone in the organization.

The person producing the information to be shared does not expect the receivers to read the information immediately. It is enough, if the receiver gets the information with a week or so.

Therefore:

Place a bulleting board in a central place in the organization, for example next to the coffee room. Anyone can post bulletins to this board and share

information. People can read the bulletin board while they are having a coffee break so information is efficiently exchanged.



One should consider what kind of bulletin board to use. Is it traditional bulletin board where paper notes can be placed with pins or whiteboard? Whiteboard could be more useful, if the bulletin board is used for drawing things and as a tool for discussions. Traditional board is more convenient when just posting information. However, the bulletin board should not be an application, if the information can not be shown in the central place in the organization.

Bulletin board should be located so that people pass it many times a day. In this way, they can easily see if there are new items on the board. Good places are for example, vicinity of coffee room or central corridor. Of course reading the posts can not still be guaranteed.

You should also think of policy how the bulletin board is used. When the items should be removed from the bulletin board and whose responsibility it is? What to post on bulletin board and what not to post (e.g. for security reasons)? Typical policies are that there is a person responsible for removing old items or the person who placed the item should remove it, when the item is not current anymore. If items out-of-date exist on bulletin board, people might lose their interest in the bulletin board soon. All notes on the board should have the posting date marked on them so that deprecated messages are easier to identify.

If there are different kind of information that should be posted to bulletin board one might consider using dedicated bulletin boards for different purposes, e.g. one bulletin board for events and one for general announcements.

For more sophisticated approach, one should see INFORMATION RADIATOR pattern, which describes creating a slideshow screen that is placed in a central place.

Information is efficiently shared in the organization as bulletin board is visible to everybody and anyone can post new items on it.

If people are working remotely or organization has multiple units, all members of the organization might not be able to see items on the bulletin board.

Event Notification Patterns for Distributed Machine Control Systems

Veli-Pekka Eloranta {firstname.lastname}@tut.fi

Department of Software Systems
Tampere University of Technology
Finland

1 Introduction

In this paper we will present three design patterns for distributed machine control system. By distributed machine control system we mean a system consisting of multiple embedded controllers which communicate with each other. These actuators control, monitor and assist the operating of a work machine or an automation process. Typically these kinds of systems have strict real-time requirements which set certain limitations for their architecture. Other key drivers of these systems are distribution, fault tolerance, safety and long life cycle.

Control systems have become large and complex systems where the software architecture plays a central role in the overall quality of the machines. However, there is not so much literature on the specific aspects of these systems. Therefore, we feel that there is a need for a pattern language to ease the burden of designing such systems.

Three patterns for this paper are a part of larger body of literature and these were selected as they are quite tightly coupled and form a whole that is part of even a bigger whole. In this paper, we will provide patlets of the referenced patterns in the pattern language to help the reader to understand these three patterns better. Other patterns in the language are not currently publicly available.

The patterns in this paper are mined during 2008-2011 in an industrial context. The companies from where the patterns were found are global manufacturers of work machines and automation systems. Initial drafts of the patterns were found during architecture evaluations in the Finnish machine industry. Then the initial versions of the patterns were given to domain experts for review. After they had reviewed the patterns, we interviewed them to gain more insights to the domain and the patterns. Finally, the current version of the patterns were written.

2 The context of the patterns

The three patterns presented in this paper are part of larger body of literature. These patterns belong to a pattern language for building software architecture of distributed machine control

systems. The whole language consists of 70 patterns and it is partially illustrated in Fig. 1. Fig. 1 shows notifications sublanguage and a few significant patterns that relate to notification sublanguage. Patterns presented in this paper are highlighted with grey background.

The semantics of an arrow pointing from pattern A to pattern B in our language is "pattern B refines pattern A". This means that if the architect has solved some design problems with pattern A, the design context is now compatible with the required context of pattern B. For example, after applying NOTIFICATIONS pattern, the designer can apply NOTIFICATION LEVELS pattern which refines the previous pattern by introducing different levels of notification messages. The designer might look at all refining patterns if there are still some unsolved problems in the context.

The root pattern of the language is CONTROL SYSTEM which describes why to have control system in the work machine. ISOLATE FUNCTIONALITIES refines CONTROL SYSTEM by separating different functionalities and placing them near where they are needed. In other words, this introduces message bus to the system. Patterns presented in this paper refine ISOLATE FUNCTIONALITIES pattern by providing means to handle and communicate events occurring in the system.

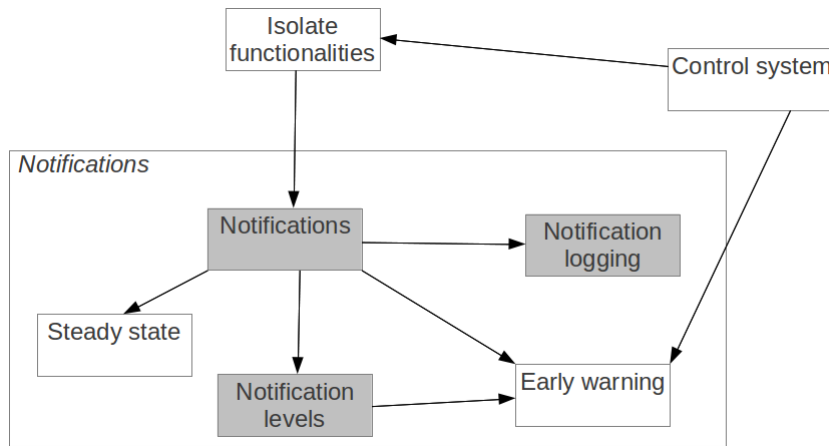


Fig. 1. Event notification patterns as a part of larger pattern language for machine control systems.

Fig. 1 shows only patterns closely related to the patterns presented in this paper. However, there are many related patterns which are referenced in this paper. These patterns are referenced in this paper using SMALL CAPS. Patterns of the patterns that are referenced in this paper are presented in Table 1.

Table 1. Patlets of the patterns that are referenced but not presented in this paper

Pattern Name	Description
CONTROL SYSTEM	How to implement a work machine that offers interoperability between systems and is highly operable with good performance? Implement control system software that controls the machine and can communicate with other machines and systems.
CONTROL SYSTEM VARIANCE	How to build a product consisting of specific software components from a product platform component library? Create a unified way to describe system configurations in a configuration file. This file is used to select the required software components and their configuration parameters for the desired system setup.
DIAGNOSTICS	How to monitor the health of the system in order to avoid surprising breakdowns? Collect such data from a system, which allows to notice if some subsystem starts to operate poorly or produces erroneous data. Usually all data values have limits where they should operate and a deviation from this indicates a risk of breakdown.
GLOBAL TIME	How to prevent different nodes on the system from getting out of sync? Use external clock, e.g GPS or atom clock, to synchronize all the nodes.
HMI	How to enable the machine operator to communicate in convenient way with the control system? Create graphical user interface where the operator can see the machine status and functions that are currently in use. Let the user to interact with the system using joysticks, buttons, etc.
HMI NOTIFICATIONS	How to notify the machine operator about the events occurring in the system? Create a way to show subset of all notifications in the HMI. Implement a dedicated service that receives notifications and shows them in HMI. This service should also be able to send notifications.
ISOLATE FUNCTIONALITIES	What is a reasonable way to create an embedded control system for a large machine? Distribute the system into subsystems according to their functionalities. Interconnect these subsystems with the bus. Use multiple interconnections between subsystems if necessary.
OPERATING MODES	In order to make sure that only the functionalities which are required can be used in current operating context, design the system so, that it consists of multiple functional modes. These modes correspond to certain operating contexts. The mode only allows usage of those operations that are sensible for its operating context.
PRIORITIZED MESSAGES	How to ensure that important messages get handled before other less important messages? The message types are prioritized according to their urgency and separate MESSAGE QUEUES are implemented for each priority.
SAFE STATE	How to minimize the possibility that operator, machine or surroundings are harmed when some part of the machine malfunctions? Design a safe state that can be entered in case of a malfunction in order to prevent the harm that machine can cause. The safe state is device and functionality dependent and it is not necessarily the same as unpowered state.
SEPARATE REAL-TIME	How to offer high-end services without violating real-time requirements? Divide the system into separate levels according to real-time requirements: e.g. machine control and machine operator level. Real-time functionalities are located on the machine control level and non real-time functionality on the machine operator level. Levels are not directly connected, they use bus or other medium to communicate with each other.
STEADY STATE	How to handle transient faults that might be generated during the system start-up? Define a time interval in which the system must reach Steady State. Being in Steady State means that the system is ready for normal operations. Before the Steady State is reached, the system can generate erroneous alarms that can be neglected.
VARIABLE MANAGER	How can you efficiently share system wide information in the distributed embedded system? For each node, add a component, which contains all the information that is relevant to operation of the corresponding node. This information is presented as state variables. The value of a variable is updated every time when a message containing the information is received.
VECTOR CLOCK FOR MESSAGES	How to find out the order of events in distributed system? Give every event a vector clock timestamp. The timestamp consist of separate message counter values for every node. The message counter of a node is updated when a message containing vector clock timestamp with larger value is received.

3 Patterns

In this section, three patterns for delivering and logging events in the distributed machine control system are presented. Patterns are presented in Alexandrian form and they contain also the known usage of the pattern.

3.1 Notifications

.. you have distributed CONTROL SYSTEM with ISOLATE FUNCTIONALITIES, so the system has a communication channel that different parts of the system can use to communicate with each other. There are a lot of messages sent through the bus. Some messages contain data and some contain control instructions for the nodes. However, you need to recognize fast when noteworthy events occur in the system. For example, when a fault occurs, the situation need to be recognized immediately so that remedying actions can be taken. There should be a way to distinguish these event related messages from other messages transfered on the bus as some of the event messages might need urgent action, e.g. entering SAFE STATE.

The nodes of the system should have a uniform way to communicate that noteworthy events have occurred in the system. The delay related to commencing safety-related actions should be minimized.

Event messages should be easily identifiable from the other traffic on the bus. In this way, quick actions can take place when an event occurs. This also makes the testing and debugging of the system easier as only event messages can be recorded for debugging purposes.

It should be relatively easy to add new events, modify or remove support for events in the system. Especially, if CONTROL SYSTEM VARIANCE is used to create a support for accessory devices , new events are likely to be added when the accessory devices are taken in use.

Events occurring in the system should be traceable. Therefore, it should be relatively easy to detect from where the event has originated. Furthermore, it should be possible to find the cause of the event easily.

Events should be delivered in fast and deterministic way in the system. Other traffic on the bus should not delay the delivery of event information even when the load of the bus is high.

In order to make the system safe and robust and at the same time easy to understand for developers, event handling should be consistent throughout the system.



Therefore:

Communicate noteworthy or alarming events and state changes in the system using notifications - a dedicated message type for notifications. Provide a way to create, handle and deliver notifications easily and enforce application developers to use these notifications.

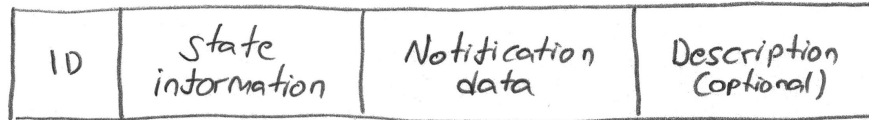


Fig. 2. Typical structure of the notification.

Deliver notifications as messages. These messages typically consist of ID field, state information, notification data and description. The typical structure is illustrated in Fig. 2. Each notification message has its own notification type identifier (ID). By this ID, the notification can be identified. For example, ID 501 could mean that oil pressure is low and ID 502 that oil pressure has reached critical level. In other words, the ID could have some semantics in it. For example, the first two (or three) numbers may pin-point the origin of the notification, i.e. which sensor or actuator caused the notification.

As mentioned earlier, notifications have states. Typically these states are "Normal" (notification off), "Active (notification on)" and "Inactive (notification on)". When the system is started, the notification state is normal, but when a notification is triggered, its state is active. Once the device (or the operator) has acknowledged that the notification was received or notification timeouts, the state of the notification is changed to inactive. If new events occur, the state is changed back to active. State diagram in Fig. 3 illustrates these state changes. When the system is restarted, inactive notifications are changed to normal state. Or there might different policies in use and only some of the notifications are changed to normal state. In some cases, it might be useful to change the state of inactive notification to normal after a timeout.

In some cases inactive notifications may become active again. For example, if the oil pressure is too low and the notification is set off but it is still active as the oil pressure is still too low, the notification should become active again after a certain time period.

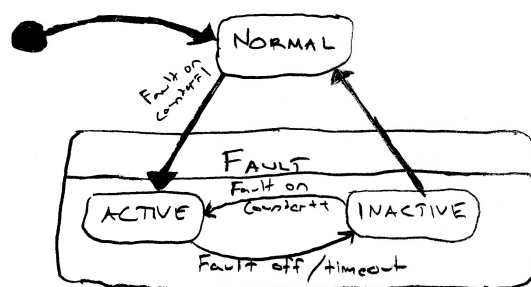


Fig. 3. States of a typical notification.

Notification data may contain additional information about the event that has happened. The notification sender attaches this to the notification message, so the receiver can use it to analyse

the situation or write it to the log file. Furthermore, if HMI NOTIFICATIONS is in use, the data can be shown to the operator of the machine. For example, notification informing that the oil pressure is too low, could contain current oil pressure value in notification data field of the message.

The description text is not typically included in the message as it would be transferred over the bus and it would increase the bus load. Therefore, description texts are mapped to notification IDs when the notification is received. The description can be then written into a log file or used in other appropriate way.

One should provide easy way for application developers to create these notifications. One might consider implementing notification service that takes care of the creating, sending and receiving of these messages. However, in many cases the processing power of nodes is not sufficient to do so. If SEPARATE REAL-TIME has been applied, one should create the notification service at least on operator level. On low end nodes, one can consider providing API or library for notification handling. If this is not possible, then the application must take care of the notifications itself.

When SEPARATE REAL-TIME has been applied, the notification service on operator level can utilize VARIABLE MANAGER. In this case, the notification service forwards notifications using VARIABLE MANAGER to components which have been configured to be interested in certain notification. For example, if node A detects a fault and creates a notification. This event is delivered to notification service which then notifies other machine control level nodes interested in this event. Fig. 4 illustrates typical structure of the system with notification service in place.

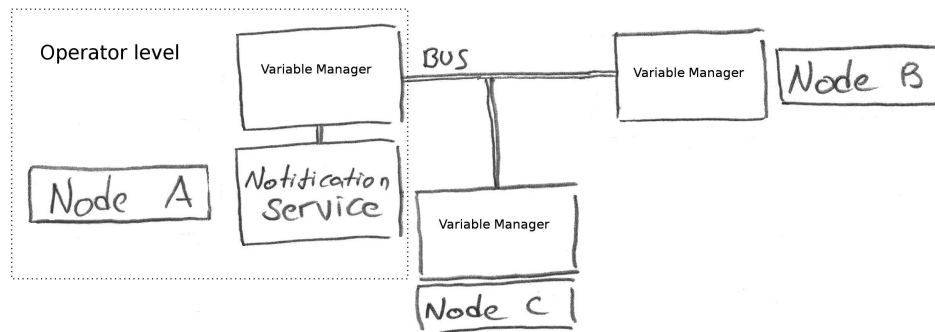


Fig. 4. Typical system structure when the notification service is used on operator level.

Using notification service eases decoupling of the application code from the notification processing. Notification service should have an interface for propagating notification, receiving a notification and setting the state of the notification. In addition, there should be a system-wide configuration file, e.g in XML that notification service reads. The configuration file contains notification IDs for each event and list of nodes which are interested in each notification. The node itself should take care of the logic when the notification is triggered and trasfered to notification service.

If there are different kind of notifications that should be delivered, one should consider applying the solution described in NOTIFICATION LEVELS. NOTIFICATION LOGGING can be used to record the history what notifications have been sent in the system. PRIORITIZED MESSAGES can be used to define priorities between notification messages and other messages. There are also times when notification occurred can be ignored, for example during start up as described in STEADY STATE pattern.

★ ★ ★

Exceptional and noteworthy events or state changes occurring in the system can be informed system wide in a unified way. It is rather easy to add, modify or remove events using the configuration file that the notification service reads.

Notifications contain the notification ID which identifies the notification type. It delivers information to the notification receiver (cause, origin, etc). Human readable description can be found by using notification ID so the human readable text does not need to be transferred over the bus. Therefore, notifications causes only minimal amount extraneous bus traffic.

Notification states make it possible to have notifications active, but in the background. This makes it possible to deliver the same notification again if the problem has not been removed. This makes the machine more reliable and easier to operate.

Notification messages can be recognized among the other traffic on the bus relatively easy. This makes it possible to process them before other messages or redirect them to separate component for processing.

Notifications in distributed system may cause some additional bus traffic as the number of sent messages increases. Therefore notifications may somewhat decrease the system performance if throughput of the bus is already a problem. However, it might be the case that the information would have to be transferred anyway.

It requires manual work to configure notifications for each node. Notification IDs also need to be documented, so it causes some extra work.

★ ★ ★

In a forest harvester, the harvester head uses notification service to inform the rest of the system about the anomalous or otherwise noteworthy events taking place in the harvester head. Marker colour (that is used to mark the cut trees) runs out. The marker's control application generates a fault and puts it in the VARIABLE MANAGER. The notification service notices this and creates a notification message containing the notification ID that corresponds to the occurred event. In addition, it adds the marker colour container current filling level to notification data and sets the notification state to "active". Then the notification is sent to bus from where the nodes interested in this information reads it. Operator level PC reads the message and informs the UI application that marker colour container empty notification has been received. The user interface application then shows a pop-up on the screen informing the machine operator about the situation. The operator clicks OK on the pop up and it is closed. The user interface application calls the interface of the notification service to set the notification

state to "inactive". The operator can continue working but the marker colour can not be used anymore. Once the marker colour container is filled up again, the node controlling the marker colour sends a notification message informing that the state of marker colour notification is now "Normal" again.

3.2 Notification levels

.. you have a distributed CONTROL SYSTEM and NOTIFICATIONS are used to communicate that something noteworthy has occurred in the system. However, there are different kinds of events occurring, e.g. operating mode changes, faults, errors and system state changes. It could be useful to have more granularity in notifications as different events have different consequences. Sometimes you need to inform the operator that a process, the machine operator has started, is completed, e.g. once the automation sequence of drill placement has completed. Furthermore, there might be a need to inform the machine operator about warnings, e.g. oil pressure low. This may not require immediate actions, but should be brought to machine operator's attention. In addition, there might be some faults, e.g. a node notices that a sensor is broken down and needs to inform this fault. The sensor fault may disable some functionality of the machine as a consequence and remedying actions in this case should be immediate. In general, different events have consequences of different severity.

Different kinds of notifications should be grouped according to their severity in order to be able to determine the consequences of the notification in deterministic way.

Different kinds of events may have different consequences. It should be easy to distinguish between these different event types and to handle all consequences of a certain event type in a uniform way. Furthermore, some event types may have need for a faster response time than other events.

Some events occurring in the system, e.g. faults, might stop some operations as a consequence. On the other hand, some events might be essential part of a normal functionality of the system. Events that can make the system stop functioning should be handled first. Generally, events should be processed in the order of severity of the consequences.

Sometimes if an event is not occurring in the system, it might be a sign of a fault or failure. The system should be able to detect if some event is missing for too long.

Operator should be able to easily see what kind of event has occurred. It should be possible to detect the event type just by taking a glance at the user interface, not reading the description of the event.



Therefore:

Attach level information to the notifications. Typical notification levels are: notices, warnings and faults. Each notification level has its own way to remedy the situation and each level has different response times.

★ ★ ★

All notification levels have their own severity and ways to react to the occurred event. "Notice" notification level consists of events that are supposed to happen in the system. For example, when the machine operator starts system self-diagnostics the process will run on its own. Now, when this process has finished a notice level notification should be triggered. This notification informs that the self-diagnostics process has finished. If HMI NOTIFICATIONS are applied, the user interface will then show this notice notification to the machine operator, otherwise nodes in the system can utilize this notification. For example, when the notification is received, it may propagate a mode change in the system, e.g from diagnostic mode to normal operating mode and nodes should make changes to their state accordingly (see OPERATING MODES for more details). "Warnings" are notifications which inform that something needing attention has occurred. For example, if oil pressure of the motor runs too low, a warning is given. "Faults" are used to inform that some part of the system is malfunctioning. For example, if boom positioning sensor has broken down, the fault notification is sent by the boom controller. Faults can make the system to enter SAFE STATE automatically when they occur or if SAFE STATE is not used, faults may make the machine otherwise inoperable. Furthermore, if HMI NOTIFICATIONS pattern is used, "faults" typically cause automatic actions from the system, even though the notification is not yet shown to the machine operator. The structure of notification message when using notification levels is depicted in Fig. 5.

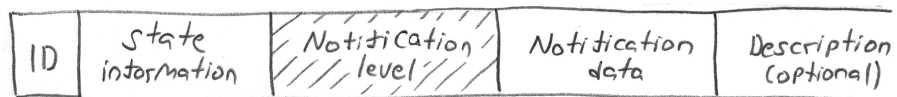


Fig. 5. Typical structure of the notification message including notification level.

Basically, the notification levels are added to the notification message as a new field. Sometimes also the first (or some other) digit of the notification ID can be used to express the notification level, e.g. 1 for notices, 2 for warnings and so on. Notification service can use this level information to determine the priority of messages. Typically faults are most urgent, then warnings and last notifications. However, one might also use different granularity for the notifications, e.g. operating event, info, warning, fault and in this case the priority may be different. All notifications have states (as described in NOTIFICATIONS), when notifications are enhanced with levels, all notification states are not utilized. For example, for notices, states "active" and "normal" are typically enough as there is no need to have other states.

Usage of notification levels is carried out through the notification service. The interface might be slightly changed to support different notification levels, i.e. adding a parameter to functions or so.

As mentioned earlier, different notification levels have different priorities. So, if multiple notifications are active simultaneously, they should be processed in the order of importance, e.g faults first, then warnings, etc. Within a single notification level, e.g. fault, it might be necessary to prioritize notifications. If this is the case, one should consider adding new notification levels.

When using notification levels it might be necessary to use PRIORITIZED MESSAGES pattern within the notification service to implement the priority order of different notification levels.

★ ★ ★

Different kinds of events can be distinguished easily using separate notification levels. This makes, for example, UI design easier as each level has its own way to be shown in the UI. Events needing urgent attention can be distinguished from the other event easily.

Notification levels enable the system to remedy a certain kind of events in their specific way. For example, when a fault occurs, the machine can always enter SAFE STATE automatically or stop all operations.

When using notification levels, it is easy to create a priority order of notifications. This is especially useful if multiple notifications are active simultaneously. However, within a notification level, more fine-grained approach might be needed.

Notification levels are relatively easy to add, modify and remove. However, implementing the processing of these levels in all nodes may require a lot of work.

Notification levels requires additional manual configuring of the system. Each level has to be defined and configured which notification resides on which level. Configuration might become quite complex, when notification levels changes between software versions.

Within a notification level more fine-grained priority order might be required. This needs additional configuring in the system and may require new field in the notification message.

Deducting notification states becomes more complex as in some notification levels all states are not necessarily used.

Bus load is slightly increased due to additional message length.

★ ★ ★

In forest harvester the machine operator activates the tree cutting by pressing a button in the hand panel. Once the machine has finished cutting the tree, the harvester head updates its status variable in VARIABLE MANAGER informing the notification service that the cutting operation has finished. Notification service creates a notification which level is "notice" as this event does not require any human intervention. The notification message is sent to the bus and the cabin computer running the user interface receives this message. Once received, the notification triggers a state change in the system as the system is now ready for the next operation. Additionally, the cabin computer lights up LED in the dashboard, informing the machine operator that (s)he can start feeding the log through the delimbing knives.

In another case, the machine operator has left the cabin door open and tries to start cutting a tree. However, once the operator presses the cut button, the cabin computer generates a warning level notification using the notification service. This notification is delivered via bus to harvester head which does not start the cutting operation. The cabin computer shows a warning to the user, that the cabin door is open and should be closed for safety reasons before the cutting operation can commence. If the user closes the warning pop up and tries again to start the cutting operation, the warning is shown again. When the door is closed, the status is

updated to normal and again a notification is generated using the notification service. Now, once the harvester head has received this notification, it changes its mode and is ready to start the tree cutting when the operator commences it.

In a third case, the machine operator has started the sawing process of a log and the saw chain breaks. Harvester head generates a fault notification and sends it to the bus. All nodes go to SAFE STATE which means they stop the current operation. A fault is shown in the user interface to the machine operator after the system has entered SAFE STATE. Now the machine operator needs to take the necessary actions to change the saw chain. Once she has done that, she must restart the system to disable the fault notification. At the start-up the machine runs diagnostics that determine if the saw is OK and as it detects that this is the case, it resets the notification state to "normal".

3.3 Notification logging

... you have distributed CONTROL SYSTEM where noteworthy or suspicious events are communicated using NOTIFICATIONS. There is a notification service implemented to handle notifications. The fast-paced real-time environment makes it hard to detect the source of a fault or error while the machine is operating. Therefore, machine operator or maintenance person needs to search for the faulty component afterwards once the machine is stopped. Furthermore, the machine manufacturer would like to gather information of typical faults for their own analysis so that the same faults typical to the model can be avoided in the next version. In addition, it would be beneficial during the yearly maintenance of the machine to find out what kind of faults and errors, i.e. NOTIFICATIONS, have occurred in the system.

How to find out later on what notifications have occurred in the system?

In many cases, when a fault or error occurs, it can not be deducted at the moment what was the root cause of the fault as the environment is too fast-paced. It might seem that a controller is faulty, but it might be caused by another controller or broken sensor. That's why it should be possible to find out later on what exactly has happened in the system. For example, when a fault is detected, the system enters SAFE STATE immediately and after a few seconds when the system has analysed the root cause of the fault, the error message is shown to the machine

The order of occurred events is important. It should be possible to reliably know the order of the events.

Sometimes it is necessary to know the exact time when an event has occurred.

Once a fault has occurred, it should be possible for the maintenance person to analyze what is the root cause of the problem that the system has reported for the operator.

Diagnostics during the yearly maintenance of the machine should be able to tell what kind of errors have happened in the system. There should be a way to produce a report of the events that have occurred in the system.

It is valuable for research and development (R&D) to know the typical faults and errors that are taking place in the system. So R&D should be able to collect statistics about the events taking place in the system. In this way, the next version of the system (software or hardware) can be developed so that these faults can be avoided.



Therefore:

Create a logging mechanism that logs notifications that occurs in the system. Add timestamps and notification source to all logged notifications. In this way, order of notifications can be deducted.

★ ★ ★

When a node sends a notification it marks that into the log file. Furthermore, the node receiving the notification should also log the notification when it receives it. A log entry should contain at least the notification ID, notification data and timestamp. Additional fields can be used whenever necessary. For example, notification levels might be important to store as well. Log entry's timestamp should contain the timestamp of the moment when the message is received.

Log entries can be kept in memory, but entries should be written into a file after a while. However, if log entries are written to a file as they emerge, it may decrease system performance. Therefore, it is advisable to keep log entries in memory and write them to a file periodically and balance between RAM usage and system performance. Additionally, this depends on the amount of notifications in the system. If there are typically only a few notifications, they probably can be written to a file as they emerge. One should also take into account the case of a sudden power failure. In that case, the log entries in memory might get lost. So it is a trade-off between reliability and traceability. Saving interval could be determined using the criticality of the availability of log entries. Most critical information should probably be written directly to a separate log file.

Using timestamps for the log entries can be troublesome. It might be the case that the nodes in the system do not share common time. If this is the case, the order of different notifications on separate nodes, might be impossible. In that case one might consider using `VECTOR CLOCK FOR MESSAGES` or `GLOBAL TIME` to solve this issue. If real-time is used to determine the order of log entries, the timestamp should use the best accuracy available.

One question is that should the logging be centralized or distributed. Centralized logging would generate bus load. Therefore distributed approach is often used. This approach is illustrated in Fig. 6. Often diagnostics tools and analysis requires that all events are gathered to a single place. This is typically carried out so that when a diagnostic service is started, it gathers all logs from nodes and merges them to a single log. This is a good option as diagnostics are usually run when the machine is not operating and therefore the bus load is not high. For further details, see `DIAGNOSTICS`. If multiple log files are merged, the original logs should be left intact and the merged log file should contain the origin of the log entry.

Over time log files can become rather large and it might take a lot of disk space to store them. That's why they should be cleared once in a while. For example, log files can be cleared once the system is updated or during the yearly maintenance. It pretty much depends on the case when this can be done. Sometimes it might be reasonable to clear log files during the reboot. Additionally, one might consider using a light-weight database system such as `SQLite` for logging.

If necessary, one might consider also filtering notifications that are logged. It might be the case that all notifications are not relevant from the logging point of view. If `NOTIFICATION LEVELS` are used, it can be easy to determine which levels should be logged and which not. It might also be a good idea to have an own log file for each notification level, so single file won't grow too large or just own log file for the most critical notifications. Furthermore, it

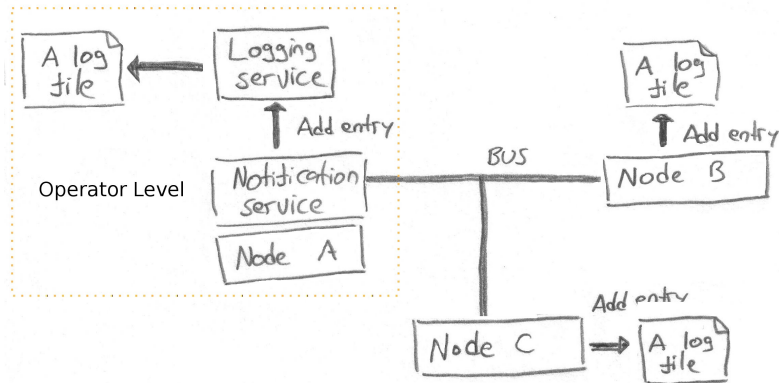


Fig. 6. Typical structure of the notification logging service.

might advisable to be able to disable filtering when necessary, e.g for debugging or maintenance purposes.

* * *

Notifications taking place in the system can be recorded in the log files for later examination. This makes it possible to create diagnostic reports later on.

The order of notifications occurred can be determined later on by gathering all log files. This makes locating the fault easier for the maintenance persons.

Notification logging may decrease system performance as a log entry has to be created for each notification sent.

May increase memory consumption of a node if multiple log entries are kept in memory before writing them to a file.

It might be impossible to implement logging service in low end nodes. If this is the case, one might need to build other mechanisms to include messages of these nodes to the log files.

* * *

In mining drill notification logging is used in each node. Intelligent drill rotation sensor notices that the drill is stuck and sends a notification to the drill controller. This notification is logged to the sensor node. The drill controller stops the drill and sends a notification forward that a fault has occurred. The system is stopped. Now the operator accesses the system diagnostics to see if the system is malfunctioning or if the drill is really stuck in the drilling hole. The diagnostics application gathers log files from all nodes and constitutes a report. From the report the operator can see that the sensor has sent a notification before the drill

controller. This indicates that the system is working correctly, because it could be the case that the drilling controller is malfunctioning and would send erroneous notifications.

4 Acknowledgements

I want to thank my colleagues Ville Reijonen, Marko Leppänen and Johannes Koskinen for their help. Especially I want to thank all industrial partners who have made it possible to mine this patterns from their systems: Metso Automation, Kone, Sandvik Mining and Construction, John Deere, Areva T&D. Thanks also to VikingPLoP 2012 participants for the valuable feedback.

Patterns for High Availability Distributed Control Systems

Johannes Koskinen

Department of Software Systems
Tampere University of Technology
Finland

{firstname.lastname}@tut.fi

1 Introduction

The patterns presented in this paper are part of a larger pattern language that is currently formed in collaboration with large global machine control companies. The patterns have been collected from the real-life systems using architecture evaluations and interviews. The previous version of the language is available in [3].

A control system is a device, or set of devices to manage, command, direct or regulate the behavior of other devices or system¹. In this paper by an embedded control system we mean a software system that controls large machines such as harvesters and mining trucks. Such systems are often tightly coupled with their environment. For example in case of a harvester, harvester head hardware needs special-purpose applications to control it. In a distributed control system, the system is divided into subsystems with each controlled by one or more controllers. These controllers are connected by networks.

2 Patterns

In this section a set of patterns from the pattern language (refer Fig. 1) is presented. The selected patterns for the paper are VOTING, STATIC RESOURCE ALLOCATION and STATIC SCHEDULING. The pattern language graph could be seen as a designer's map for solving design problems. The design process begins so that the first pattern to be considered is CONTROL SYSTEM in the middle of the graph. After the designer has made the design decision to use the pattern, she may follow the arrows to the next patterns. An arrow means that the following pattern can be applied in the context of the resulting design from the old pattern. In other words, the subsequent pattern refines the design. However, a single pattern may be used regardless of the usage of previous patterns if the context of the pattern matches the current design situation.

The patlets for the patterns as well as the referenced patterns not included to this paper are presented in Table 1.

¹ http://en.wikipedia.org/wiki/Control_system

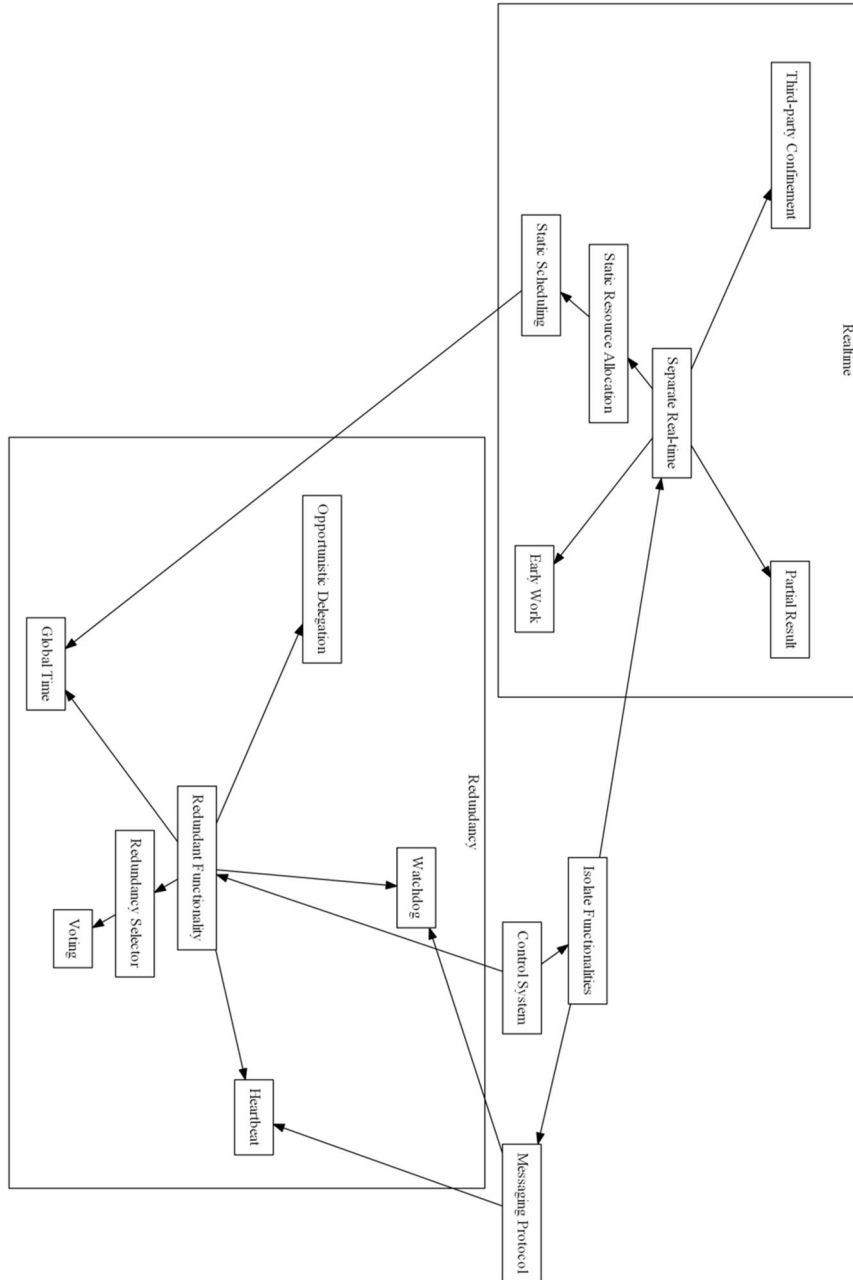


Fig. 1. Part of the pattern language for distributed control systems.

Table 1. Patlets for the included and referenced patterns.

Pattern	Patlet
Control System	<p>How to implement a work machine that offers interoperability between systems and is highly operable with good performance?</p> <p>Therefore:</p> <p>Implement control system software that controls the machine and can communicate with other machines and systems.</p>
Global Time	<p>How to prevent different nodes on the system from getting out of sync?</p> <p>Therefore:</p> <p>Use external clock, e.g GPS or atom clock, to synchronize all the nodes.</p>
Early Work	<p>How to execute resource-greedy tasks which may require more resources than are available after normal operating mode is established?</p> <p>Therefore:</p> <p>The processes should be prepared (like pre-calculating values to be used later on) in system startup, because there are usually easier to dedicate CPU time, memory, etc for heavy calculations than later on.</p>
Error Counter	<p>How to distinguish substantial faults from false alarms or transient faults?</p> <p>Therefore:</p> <p>Create a counter which threshold can be set to certain value. Once the threshold is met, an error is triggered. The error counter is increased every time a fault is reported. The counter is decreased or reseted after certain time from the last fault report has elapsed.</p>

Limp Home	<p>When a part of the machine is malfunctioning, how to still operate the machine to some extent? For example, to drive the machine from forest to the nearest road.</p> <p>Therefore:</p> <p>Divide the sensors and actuators into groups according to the high level functionalities, such as drive train, boom operations etc.. The groups may overlap. Malfunctioning device only disables the groups it belongs to and the other groups remain operable.</p>
Notifications	<p>How to inform operator or communicate to subsystems that something worth of noticing has occurred in the control system?</p> <p>Therefore:</p> <p>Communicate noteworthy or suspicious state changes in the system using notifications. Implement also Notification service that is used to create, handle and deliver notifications.</p>
Redundant Functionality	<p>How to ensure availability of a functionality even if the unit providing it breaks down or crashes?</p> <p>Therefore:</p> <p>Clone the unit controlling a critical functionality. One of the units is active and other is in a hot standby mode. If the controlling unit fails, hot standby mode unit takes over controlling the functionality.</p>
Redundancy Selector	<p>How to remedy the failure of a redundant unit when activation time of the hot standby unit is too long?</p> <p>Therefore:</p> <p>Design the system so that all redundant units are active at the same time. Add a monitoring component that takes outputs of all redundant controlling units as inputs and examines the output and chooses which unit's output is forwarded as a control signal.</p>

Separate Real-time	<p>How to offer high-end services without violating real-time requirements?</p> <p>Therefore:</p> <p>Divide the system into separate levels according to real-time requirements: e.g. machine control and machine operator level. Real-time functionalities are located on the machine control level and non real-time functionality on the machine operator level. Levels are not directly connected, they use bus or other medium to communicate with each other.</p>
Start-up Graph	<p>How to determine an optimal start-up sequence when the system setup may vary?</p> <p>Therefore:</p> <p>Design the start-up graph of devices and their dependencies, start-up times and resources requirements during start-up and during normal use. The start-up monitor component determines the system setup and constructs the correct system start-up sequence.</p>
Static Resource Allocation	<p>How to make sure that the critical services will always have the resources needed?</p> <p>Therefore:</p> <p>All the resources needed for critical services should be pre-allocated during the system startup. The resources are never deallocated afterwards (i.e. the resources are fixed for the service).</p>
Static Scheduling	<p>How to schedule real-time processes efficiently with low overhead?</p> <p>Therefore:</p> <p>Divide the system into executable blocks. The executable blocks can be e.g. applications, functions, or code blocks. Scheduler has time slots of fixed lengths. The developer or compiler divides the blocks into these time slots, thus scheduling the program statically.</p>

Watchdog	<p>How to make sure that a node crash does not go unnoticed?</p> <p>Therefore:</p> <p>Add a watchdog component to each node to monitor the behavior of the application(s). If the application does not react within a given time limit, the watchdog deems the node to malfunction and starts remedying actions, such as reboot.</p>
Voting	<p>How to make reliable decisions in a distributed system where high correctness is required?</p> <p>Therefore:</p> <p>Create redundant nodes calculating the control output.</p>

2.1 Voting

...there is a CONTROL SYSTEM with high availability and correctness requirements. However, there might be situations, when the functionality of a node may fail. REDUNDANT FUNCTIONALITY pattern is used to ensure high availability of functionality even in case of broken node. Still, we need to ensure correctness of the decision made by the control nodes. For example, a failure in a single sensor should not affect a node's output value.

Reliable decisions should be made in a distributed system where high correctness is required.

The value of the control output should not be affected if a node providing it malfunctions or crashes.

There is no time available for off-line fault diagnosis and recovery actions, the other units should be used to mask and detect a fault in one of the hardware unit.

Decisions should be based always on the correct information. Potentially dangerous situations may occur if the information is not correct due to the failure in a node or a sensor. Usually, a fault tolerance in a control system is achieved through redundancy in hardware.

It should be possible to add redundant units later on, for example, if the requirements for the correctness are changed. In addition, the system should be scaled up and down depending on the current target usage.

Therefore:

Create redundant nodes calculating the control output.

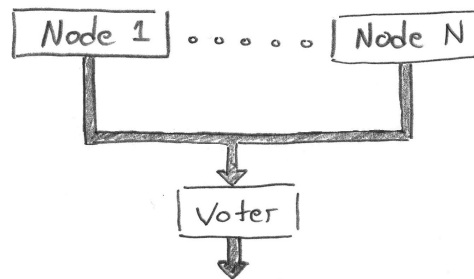


Moreover, an additional unit is added to the system to collect the output from these redundant nodes. This unit is called the collecting node (Voter).

The outputs of the nodes must be comparable. The collecting node then deducts which output value has got the most "votes" and gives that forward as control output of the (sub)system. Simple disagreement of the output indicates a fault, but cannot identify the faulty node nor the correct output.

To achieve continuous operation and correct output, the majority vote of the outputs of three or more identical nodes is used. Simple two out of three votes masks single-node failure. The more redundant units are used in the system the more failing nodes can be masked. For example, to mask simultaneous failure in two nodes, five nodes are needed altogether.

The collecting unit accepts all N control outputs as inputs and uses these to determine the correct, or best, output if one exists. In addition, if the faulty node can be identified, NOTIFICATIONS or ERROR COUNTER can be used to inform the rest of the system about the node.



A significant problem with voting is that the output values must be comparable. With switches this is easy as the switch output value is either open or closed. On the other hand, with analog signals, an input of the Voter can be regarded as to be correct if the input value is close enough with others. The output of the Voter is then generated based on the inputs, for example, using average of the correct inputs.

If an input of a node is incorrect, the output is also incorrect, even if the node is fully functional. Thus, the sensors should also be replicated to get correct inputs for the redundant units. It is also possible to have different versions of the software in each unit, masking failures in software. Moreover, the Voter can take the reliability of each node into consideration and determine the most likely correct answer. However, this will require knowledge of each node and/or additional acceptance or self-checking tests. For example, it might be known that the output of the value should be within a predetermined range.

The voter component creates a single point of failure and WATCHDOG should be used to monitor a voter node. LIMP HOME can be used to select an output value from one of the redundant nodes if the voter node is broken. REDUNDANCY SELECTOR pattern can be used instead of this if there is no need for voted values. The selector designates one of the input values as an output value. The selection criterion is simpler than with voting, for example, the first input with a particular value is always selected.



Failure in one unit will not cause an erroneous output signal. More than one node is generating the output signal and faulty values are left out by the voting mechanism. In addition, because of the redundancy in the system, the output is always available, even if one of the nodes fails.

Self-checks can be used to improve the voting process. If the nodes indicate a failure in their software or components, the information can be taken into consideration when the output value is decided.

With voter, it is also possible to detect the malfunction in a single unit and replace the broken unit by a good one.

On the other hand, redundant units and sensors increase system costs and take more space. Synchronization of the units may be difficult and communication between redundant units increases complexity of the system.

As the voter does not communicate directly with the nodes, it does not affect the operation of active unit. However, the voter component creates a single point of failure. To be reliable, the voter component should be robust and simple enough.



For example, a machine control system has its own vehicle identification number (VIM) 123. The identification number of the system is stored to the nodes. After service operation, a node from some other machine (with VIM 234) is moved to our machine. The node contains still the vehicle identification number 234, when the system is started up. During startup, the voting procedure is used to identify the current VIM of the machine. Each node tells its understanding of the machine's VIM and the VIM that gets the majority of the votes (in this case 123) is selected. The identification number is updated on all the nodes.

2.2 Static Scheduling

...there is a CONTROL SYSTEM where SEPARATE REAL-TIME has been used to divide the functionality to high end functionality and real-time control functionality. With scheduling, the processes are assigned to run on the available CPUs, since typically the number of processes exceeds the number of available CPUs. The processes have strict real-time constraints with hard deadlines. Execution times of the processes and timing constraints are known beforehand. In real-time systems, a process response time to a certain event must be met, regardless of system load, or else the execution of the process fails. In other words, real-time computations have failed if they are not completed before their deadline. In case of hard deadlines, missing the deadline will cause the whole system to fail.

Real-time processes should be scheduled safely with low overhead.

With limited processor power in embedded systems, the scheduling process should be lightweight with very little run-time overhead. Selecting the next process to run should cause as little run-time overhead as possible.

The most important requirement of a hard real-time system is predictability as missing the deadline will cause the whole system to fail.

The number of the real-time processes is known beforehand.

Therefore:

Divide the system into executable blocks. The executable blocks may be e.g. applications, functions, or code blocks. The scheduler has time slots of fixed lengths. The developer or compiler divides the blocks into these time slots, thus scheduling the program statically.



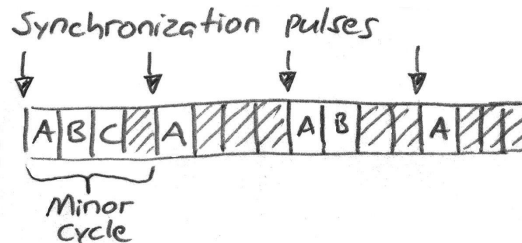
If the timing properties of all processes are known a priori, the schedule of the processes can be built statically during compile time based on the prior knowledge of execution times, precedence and mutual exclusion constraints, and deadlines. Once a schedule is made, it cannot be changed in run-time. On the other hand, run-time overhead of scheduling is very small as the scheduling decisions are made beforehand. In addition, when the scheduling is done based on worst-time estimations of the execution times, the processes cannot be overrun. The compiler should give an error message, if the scheduling is not possible.

The processes are divided into executable blocks, with a dedicated execution period time (e.g. every 10 ms). The schedule has a main cycle (e.g. 100 ms), which is divided into minor cycles (e.g. 10 ms). The period time tells how often the block is periodically executed and it must be a multiple of the minor cycle time. The maximum of the period time is main cycle time.

To schedule processes, the compiler maps the blocks onto minor cycles, which constitute the complete schedule (i.e. main cycle). In this way, each minor cycle executes code from the execution blocks based on their period time. In its simplest form, each minor cycle is just a sequence of procedure calls.

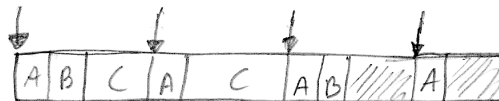
The mapping is done by calculating the worst-case execution times for every execution block using prior knowledge of the execution times for the operations carried out in the block. The scheduling is synchronized using synchronization pulse (like external clock interrupt), which starts each minor cycle.

For example, let us have processes divided into execution blocks A, B, and C. Block A has a period time 40 ms, B 80 ms, and C 160 ms. The main cycle is 160 ms and it is divided into four 40 ms minor cycles. Thus, the first minor cycle will have code from all the blocks and rest of the minor cycles contains code from the blocks depending on their period time. As the block A has period time of 40 ms, its code is executed in each minor cycle. Block B has code in every second minor cycle and C only in the first cycle. If all the CPU time is reserved for the execution blocks, there cannot be other external interrupts except synchronization pulse, but the CPU-usage can be very high.



If the system should have interrupts, the worst-case execution time for the interrupt handling should be known beforehand and for each minor slot, the corresponding free time for the interrupt handling should be reserved. Alternatively, the interrupt can set a flag and the actual interrupt processing can be done periodically in pre-reserved time slot.

If the scheduling is preemptive (i.e. one process can interrupt the other) and based on priorities, the execution time of a execution block or process can exceed one minor cycle. In this case, the processes with shorter period time have higher priority and they will interrupt other processes. For example, if the execution time of the block C is 50 ms, it will be distributed into two minor cycles. In the beginning of the second minor cycle, process A interrupts process C and it is executed first.



In similar way, it is possible to schedule dynamic, non time-critical processes in addition to time-critical, statically scheduled processes. The static scheduler is run in a single high priority task, which preempts the lower priority dynamic tasks. In each minor cycle, there should always be some time available for dynamic processes assuming that no interrupts has been occurred. On the other hand, CPU utilization is usually worse than pure static scheduling.

One can use GLOBAL TIME to synchronize execution between multiple units for increased accuracy.



When the processes are scheduled statically, run-time overhead is limited as all the scheduling decisions are done beforehand. In addition, there is no need to make preparations for unsuccessful scheduling, as there should always be enough CPU time.

With the scheduling is based on worst-case execution times, the process will never overrun. In addition, scheduling is predictable as the time slots are fixed and the number of the process is always same.

On the other hand, with pure static scheduling, dynamic processes cannot be created during run-time. However, high CPU utilization is easy to achieve, especially if there are no interrupts.

A process will need to split into a number of fixed sized procedures. This may be error-prone. In addition, cycle times set limits to calculation and period times.

The pattern can also be used to document the system. The proposed scheduling is visualized as a graph to give information on when each process is executed.

It might also be impossible to calculate execution times for operations in all cases. If the worst-case estimations are used, it will lead lower CPU utilizations.



Processes are scheduled statically in a power plant control system. For each process, the total execution time is calculated based on execution time of each instruction and measured timing information for operations. When the processes are compiled, the scheduling is defined and the code of the processes is divided into execution blocks, which worst-case execution time is the same as the minor cycle time. Each minor cycle executes code from one block. During one major cycle, all the blocks get executed. In this way, all the processor time is used to execute processes and CPU utilization is very high.

2.3 Static Resource Allocation

...there is a CONTROL SYSTEM with a node having several processes. At least some of these processes provide critical services that are essential for system functionality. Such a service should always be available and could be e.g. emergency message handling, which sends an emergency message to the main node via bus in case of failure. Critical services usually background processes, which are triggered by a certain event like a failure in the system, thus making the exact execution moment unpredictable. However, the critical services should always be available. In addition, these kind of services usually have strict real-time response time constraints and thus the real-time part is separated with SEPARATE REAL-TIME.

The critical services should always have the resources needed.

In embedded control systems, there is usually limited amount of resources (like memory, bus bandwidth, processing power) available to be shared for all the processes. Still, there must always be required resources for the critical services.

The resources required by the critical services should be available immediately and there might not be time for waiting other processes to free the desired resources. As critical services tend to be triggered by an event, there is little or no time for resource allocations or initializations.

Nondeterministic timing of the dynamic resource allocation makes it hard or even impossible for a service to meet strict real-time constraints as the allocation can take up more than the available time required for the service.

Therefore:

Pre-allocate all the resources needed for critical services during the system startup. The resources are never deallocated afterwards (i.e. the resources are fixed for the service).



Basically, static resource allocation is easy to implement. The critical services are started using START-UP QUEUE or START-UP GRAPH. All necessary allocations and initializations can be carried out during system startup, for example, using EARLY WORK or constructing a special boot image with pre-allocated memory regions. When entering in normal operating mode, the services will not allocate any additional resources nor will they free any pre-allocated ones.

It is very important that static resource usage is kept to an absolute minimum. EARLY WORK can be used to decrease statically allocated resources if all the preparations are carried out during system startup and only the resources required by the core functionality of the service are statically allocated.

Larger services should be divided into two smaller ones, whenever possible. One of the parts will provide the critical service with static resource allocation and the other containing rest of the service. The latter part will not require any fixed resources for its functionality. This is even more important as resources required by more than one service cannot be reserved statically just for one critical service. For example, message handling service should be divided into two new services, one for emergency messages and the other for regular messages. As the emergency message service should always be available, all its resources (such its own messaging slots and queues) are pre-allocated.

To statically allocate CPU-time for the critical services, one may use STATIC SCHEDULING to share processor time statically for each service. To allocate memory statically, FIXED ALLOCATION [1] or STATIC ALLOCATION PATTERN [2] can be used.



When all required resources are fixed for a critical service, it can never run out of resources.

As the resources allocated by the critical services are not available for the other services, fixed resource allocation means usually increased resource requirements.

The response times are faster as the service need not to wait for the resources to be deallocated by other processes.

Allocating resources statically also increases speed of the critical services as the allocation is done in the system startup. When the cost of having additional resources (such as memory) is relatively small, this pattern can be used for all services to increase the speed of the whole system. In addition to speed, predictability of the run-time execution also goes up as the nondeterministic timing of the resource allocation can be left out.



In a control system, nodes send emergency messages (i.e. EMCY messages) to other nodes via bus in case of fatal error. The bus capacity is divided into time slots, each containing one message. Before a message is sent, a sender allocates one slot for the message. To ensure messaging capacity for EMCY messages, one slot is statically reserved for critical messages. As the slot is used only for critical messages, it is always available and immediately ready to use in case of fatal errors.

3 Acknowledgements

I want to thank my colleagues Ville Reijonen, Marko Leppänen and Veli-Pekka Eloranta for their help. In addition, I want to thank all industrial partners for their valuable cooperation in our pattern mining process: Metso Automation, Kone, Sandvik Mining and Construction, John Deere, Areva T&D. Especially, I would like to thank our writers' workshop group for new ideas and comments.

4 References

- [1] Noble, J., Weir, C.: Small Memory Software: Patterns for Systems with Limited Memory. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
- [2] Douglass, B.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley Professional, USA (2003)
- [3] Eloranta, V-P., Koskinen, J., Leppänen, M. and Reijonen, V.: A Pattern Language for Distributed Machine Control Systems, ISBN 978-952-15-2319-9, Tampere University of Technology, Department of Software Systems. Report, vol. 9, Tampere University of Technology, pp. 108, 2010.

Patterns for Distributed Machine Control System Fault Tolerance Modes

Marko Leppänen
{firstname.lastname}@tut.fi

Department of Software Systems
Tampere University of Technology
Finland

1 Introduction

In this paper we will present four patterns for fault tolerance modes in distributed machine control systems. A distributed machine control system is a software entity that is specifically designed to control a certain hardware system. This special hardware is a part of a work machine, which can be a forest harvester, a drilling machine, elevator system etc. or some process automation system. Some of the key attributes of such software systems are their close relation to the hardware, strict real-time requirements, functional safety, fault tolerance, high availability and long life cycle.

Distribution plays a major part in the control systems. Different functional hardware parts of the machine are physically apart from each other and their corresponding control software is usually located in a embedded controller node near the controlled hardware. The nodes must communicate with each other in order to perform their functionalities. It is also common that the system nodes have very wide variety in their computational capabilities. Usually the system has several simple embedded controllers with limited computational abilities also known as low-end nodes. These nodes use sensors to gather information from the outside world and use actuators (e.g. hydraulic valves) to perform acts upon the environment. In addition to these embedded controllers the system may contain one high-end node that has processing power that is comparable to a common desktop PC. Due to these facts, the design of a distributed control system is usually very mode-based. This means that the system varying use cases are usually implemented as separate modes of the software. The mode-based behavior of such systems is discussed in these patterns in more detail.

The patterns in this paper were collected during years 2008-2011 in collaboration with industrial partners. Real products by these companies were inspected during architectural evaluations and whenever a pattern idea was recognized, the initial pattern drafts were written down. These draft patterns were then reviewed by industrial experts, who had design experience from such systems. After these additional insights, and iterative repetitions of the previous phases, the current patterns were written down. We hope that the final pattern language can be tested on implementation of some real system after all patterns in the language are published.

The published patterns are a part of a larger body of literature, which is not yet publicly available. A small subset has been previously published as [1]. All these patterns together form a pattern language, which consists of more than 70 patterns at the moment. A part of the pattern language in this paper is presented in a pattern graph (Fig. 1) to give reader an idea of

how these selected patterns fit in the language. These four patterns are closely related in the pattern language and therefore are ideal to be submitted together as a whole. In the following sections, all the pattern names are written in SMALL CAPS.

In the second section, we will first introduce our pattern language and the pattern format. Following this, the selected four patterns are presented in detail. Finally, the last sections contain the acknowledgments and references.

2 Patterns

In this section, a set of four patterns is presented. Together, these patterns form a sublanguage in the pattern language in Fig. 1. The pattern graph is read so, that a pattern is presented as a box in the graph and an arrow presents a connection between the patterns. The connection means that the pattern from which the arrow emerges is refined by the pattern that the arrow points to. In other words, if the designed system still has some unresolved problems even after some pattern is applied, the designer can look to the refining patterns for yet another solution if they want solve the current design issues. The patterns refine each other extending the original design with other solutions.

The patterns refine each other extending original design with more elaborate solutions. The CONTROL SYSTEM pattern which is the root of this branch and is referenced in the following patterns. The CONTROL SYSTEM is the central pattern in designing these systems. It presents the first design problem the system architect will face: Is a control system needed in this context? The pattern is introduced in patlet form in Table 1.

Our pattern format closely follows the widely-known Alexandrian format [2]. First we present the context for the problem. Then, the problem is concentrated in a couple of sentences that are printed with a bold font face. After that, a short discussion about all forces that are affecting the problem is given. In a way, it is a list of things to consider when solving this problem. Then, after word "Therefore:" the quick summarization of the solution is given. Then, after a three star transition line, the solution is discussed in a detail. This section should answer all the forces that were left open in the previous section. Then an another star transition marks the end of the section. This section describes briefly the consequences of applying this pattern. After the last star transition a real life example of the usage of this pattern is given.

Table 1: Patlets

Pattern Name	Description
CONTROL SYSTEM	Implement control system software that controls the machine and can communicate with other machines and systems.
OPERATING MODE	Design system so, that it consists of multiple functional modes. These modes correspond to certain operating contexts. The mode only allows usage of those operations that are sensible for its operating context.
LIMP HOME	Divide the sensors and actuators into groups according to the high level functionalities, such as drive train, boom operations etc.. The groups may overlap. Malfunctioning device only disables the groups it belongs to and the other groups remain operable.
SAFE STATE	Design a safe state that can be entered in case of a malfunction in order to prevent the harm that machine can cause. The safe state is device and functionality dependent and it is not necessarily the same as unpowered state.
SENSOR BY-PASS	Implement a mechanism that the value provided by a sensor can be replaced with a default or simulated value.

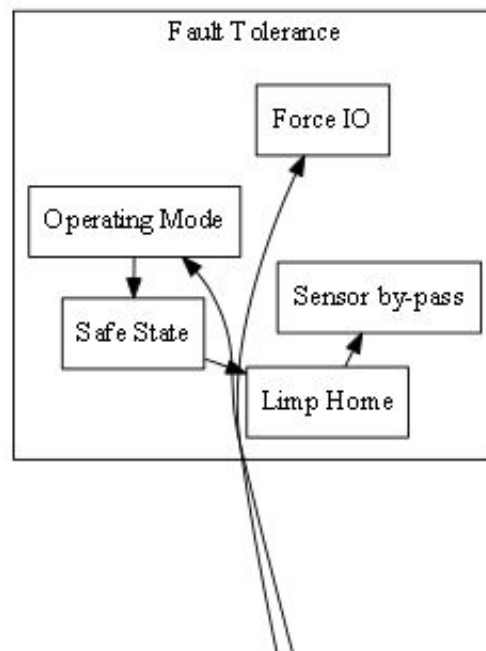
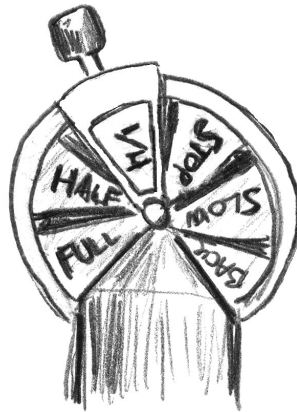


Fig. 1. A part of the pattern language for distributed machine control systems in a graph form.

2.1 Operating Mode

...there is a distributed CONTROL SYSTEM, that is used for productive work. However, there are multiple situations when the system is not in active use, but is rather under some maintenance operations or has encountered a fault. Thus, the operating context is different and the machine functions must be handled somewhat differently than in the normal operation. Usually the different operating contexts mean that some functions should be limited or even fully disabled. For example, during an update, all actuators should be turned off for safety reasons and only the bus and nodes should be active and the main user interface shows information about the update progress.

Only the functionalities which are required and necessary should be able to be used in the current operating



The control system is a complex entity which has multiple different uses and functions. These arise from the fact that the system has a plethora of different, often even contradicting requirements. The system may even have many different user roles. Therefore, the system may have different use cases which should be handled without possibly dangerous consequences. For example, a harvester is used to cut down trees in the forest when the harvester head malfunctions. When a maintenance person arrives to the site, it should be easy to inspect the work machine so that all available information is at hand. If the maintenance person wishes to test the system, it should not cause risky situations to the operator or the environment.

For a developer, it is cumbersome to build functions that depend on many different flags or conditions. For example, if all the system functions must ask if the parking brake is engaged, the bus load becomes too high and real-time requirements are more difficult to fulfill.

Therefore:

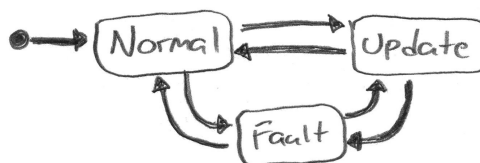
Design system so, that it consists of multiple functional modes. These modes correspond to certain operating contexts. The mode only allows usage of those operations that are sensible for its operating context.

* * *

There may be a mode for normal operation, maintenance mode, fault handling mode, update mode, parking mode and so on. These modes may be nested in a hierarchical way. Every mode may have some submodes, but it is important to keep the hierarchy simple enough. A good rule of thumb would be to use only three levels of mode nesting. Typically the default mode is one of the highest level modes which is entered when the system starts up and all other modes need some special action to be entered.

Modes can be designed for completely different use cases (maintenance mode vs. normal mode) or they can implement some functionality. For example, if the parking brake is engaged, it may trigger a mode change that affects many other functions. It is easier to design the subsystems so that they detect the global mode and use this information to deduce if some action is allowed.

Fault handling must be thought out in early stages of the system design. It is very hard to add fault handling afterwards, when the business logic is already designed and implemented. This can be remedied by using a separate fault handling mode, that is entered when a fault is encountered. In this way, the normal operation is halted until the failure is cleared out.



Modes have a strong connection to state chart states, so the system designer may draw some state diagrams when documenting and designing the system modality. There are even some pragmatic approaches that rely fully on the state paradigm when designing and implementing software systems [3]. These approaches suit well systems which have strong modality. Distributed machine control systems often exhibit these attributes.

* * *

The system's main functionality is easier to understand as the modes isolate the different usages in a compact way and it is easy to validate if the system is prepared for exceptional use cases.

The modes encourage piece-meal growth of the system architecture as the main functionality can be implemented separate from the more elaborate use cases. However, there are some functionalities which are needed to be implemented in multiple modes. This may clutter the design as the module may have to check in which mode it is used in.

* * *

A harvester has a parking brake that prevents it from accidentally moving. When the parking brake is engaged, the harvester can not move, but on the other hand, opening the cabin door does not cause an alarm as it would normally do. The parking brake functionality is implemented as a submode to the normal operating mode. Thus all the modules may just check the mode and perform their operations accordingly.

2.2 Limp Home

...there is a distributed machine CONTROL SYSTEM which consists of several semi-independent parts. These parts may be for example sensors and actuators, buses, controllers and computers. Some parts are just hardware and some parts include software systems. The full functionality mode depends on the services provided by all these separate parts. However, many of these parts exist only for some highly specialized functions. All hardware are prone to malfunction, but it would be useful if the whole machine is not incapacitated if something less crucial breaks. The system has employed OPERATING MODE pattern to enable state-based behavior.

When a part of the machine is malfunctioning, the machine should be operable to some extent. For example, to drive the machine from forest to the nearest road.

It is not sensible to incapacitate the whole system if some optional system malfunctions. If a part of the systems exists only to make the working environment more pleasant or to make little enhancements to the productivity, the operator might want to keep up working even if this kind of functionality is lost. In addition, the maintenance facilities may be far away from the machine. It would be useful to use the machine to some degree until a prescheduled maintenance break or until the machine would in any case leave the work site.

There are some functions which affect profoundly the overall safety of the system. These functions should be kept working as long as possible in any situation.

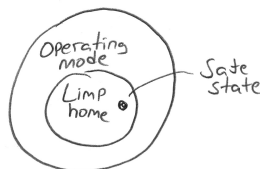
Therefore:

Divide the sensors and actuators into groups according to the high level functionalities, such as drive train, boom operations etc.. The groups may overlap. Malfunctioning device only disables the groups it belongs to and the other groups remain operable.

* * *

The system architect should design the system so, that there is a separate core functionality that is possible to perform under most situations. The core functionality might be for example just the basic movement of the machine. This core functionality is then extended with the more sophisticated services. All additional hardware that is now required is grouped optional. If some hardware in this group malfunctions, only the affected part is disabled and the driver can continue in the degraded mode. A group of devices that are required for a certain service may overlap with other services. If this is the case, if some overlapping device malfunctions, both services are incapacitated. If a device is only in one group, it will incapacitate only this service when a malfunction occurs.

There might be some services where the malfunctioning device only affects some high level functionality and when this kind of device breaks down, the system operator must do more to remedy the situation. If this is the case, the system productivity may be diminished, but the operations may otherwise continue normally. It is crucial for the system designer to identify these kinds of devices and make sure that the manual operation mode is possible.



The system should be fully inoperable only if the operation would severely harm the machine or safety of the people. Even then, some fully manual operating mode should be allowed for unforeseen situations, such as salvaging the machine from a hazardous situation (for example, driving the drill out of a collapsing mine).

The designer should try to make sure that if a subsystem breaks down, it will not lock the rest of the system up. For example, a broken down boom that is attached to some work item should be able to let the object loose in order to ensure that the work machine could still be driven out of the work site.

SAFE STATE is an approach for more severe failures.

* * *

After the devices are grouped according their criticality, the user may still make the decision to operate the system even if some non-critical system is failing. Thus, the system can still produce some value even when some parts of the machine are incapacitated.

In addition, the system is easier to test as the core functionality is clearly separated from the higher level services. However, in some cases, the breaking down of a subsystem may still incapacitate the whole system.

* * *

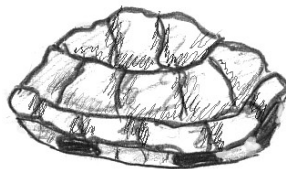
A drill machine has core functionality that is implemented using low-end controllers. More elaborate services are provided by a cabin PC. These services include work maps, productivity tracking, fleet control etc.. If the cabin PC breaks down, the machine is still able to drill and drive around the work site even though the high level supervisor services are down.

2.3 Safe State

...there is a distributed machine CONTROL SYSTEM, which consists of several semi-independent parts. These parts may be for example sensors and actuators, buses, controllers and computers. All hardware may break and software systems may contain bugs that cause the system malfunction. Malfunctioning control system acts erratically and may cause severe harm to itself, surrounding environment and/or operator or other people nearby. The system has employed OPERATING MODES pattern to enable state-based behavior.

The possibility that operator, machine or surroundings are harmed when some part of the machine malfunctions, should be minimized.

The system should never cause harm to environment or people, even during a malfunction. This might be difficult to ensure as the malfunctions can cause unpredictable behavior.



Therefore:

Design a safe state that can be entered in case of a malfunction in order to prevent machine from causing harm. The safe state is device and functionality dependent and it is not necessarily the same as unpowered state.

* * *

There is only one safe state which is defined according to the system's requirements in its operating environment. When a predefined fault occurs, the malfunctioning system performs some last preparations and enters the safe state. The system will not re-enter the normal operating mode until the failure is corrected and some resetting action is made. This mode is therefore reserved for the most severe failures as the system can not give any more added value to the user and it is not possible to use it anymore for productive work. In a complicated multi-node system it is however possible to just for some of the nodes to enter the safe state and some other subsystems can still function. For example, when a low level machine control system fails, the higher level monitoring systems may still function and give additional information about the failure.

It would be best to design the safe state so, that the system becomes automatically more safe if some parts are incapacitated. For example, in a modern nuclear reactor, the magnetically attached control rods are automatically dropped to the reactor core when the control system fails to keep them up. In this way, the nuclear fission stops in the case the control system is not working properly.

LIMP HOME is an alternative approach for failures that are not so severe. It allows some simple functionality to be still carried on regardless of a failure in the system. LIMP HOME can only be used in situations where a clearly separate and well-defined part of the system is malfunctioning. In that case, it is possible to continue the usage of rest of the system so that it does not present a threat to safety.

* * *

The system is entering a predetermined safe state in the case of a malfunction. This safe state usually prevents any further harm to the environment, the operator or the system itself. However, the safe state might not be easy to determine as the machine may face some unforeseen situations.

* * *

An industrial bus connects process monitoring and controlling I/O cards together with higher level services that are located in the control room. The control room PC polls the cards with a fixed interval and the I/O card keeps track if it is read from a higher level device. Thus, the reading of the card acts as a Heartbeat. If the polling ceases for a certain time, the I/O card decides that the upper level has crashed and enters a safe state to prevent further harm. This safe state is heavily device-dependent and may affect other parts of the system as well. For example, if the card controls pumps that moves cooling fluids in the pipes, the safest action is to stop the process that needs cooling as stopping only the cooling pump would cause overheating and further damage.

2.4 Sensor By-pass

...the distributed CONTROL SYSTEM uses some advanced algorithms to control the hardware. These algorithms need some additional sensors that are in essence additional hardware - thus possible malfunctioning parts that are complicating the system design. However, some of these sensors are not crucial for the algorithm, but give only additional precision or some other advantage to the algorithm.

An advanced control mechanism should still be operable even when some sensor of minor importance is faulty.

Some sensors are incorporated into the system design only to give small advantages in productivity or the user experience. However, they are prone to malfunction, as any hardware is. The system productivity should not decline any more than absolutely necessary, if some fine-tuning sensors break.

Therefore:

Implement a mechanism that the value provided by a sensor can be replaced with a default or simulated value.

* * *

If a sensor is detected to be working improperly, there are two options. The system can cease to use the full functionality provided by this service (DEGRADED STATE) or the value it is providing is corrected with an estimate. In some cases, a simpler version of the sensor could also be used, if such a sensor is present in the system. For example, a GPS-based leveling sensor might be replaced with an analog bubble level sensor. If the values are estimated, the estimate could be the last known good value, some extra-/interpolation from it, a constant value or even a simulated value calculated from other inputs. In this way, the full functionality is still retained, but at the cost of a lower precision.

* * *

The system can still use some high level algorithms, although with lesser precision. This means that even if an sensor breaks, the system is still operable to a high degree. If the system has multiple ways of producing a sensor value, an alternative way to deduce the value can be used and thus higher precision may be achieved.

However, it is often difficult or impossible to give reliable estimates of a value if a sensor has broken down. Usually the sensor is such that when it breaks down, no information is available from that part of the machine. Then the control algorithm has basically to guess the values the sensors would be outputting.

* * *

A forest harvester controller operating platform supports forcing values to each inputs. This mechanism allows to maintenance personnel to isolate if a sensor is faulty.

3 Acknowledgments

I wish to especially thank my colleagues Veli-Pekka Eloranta, Johannes Koskinen and Ville Reijonen for their valuable help and input during the gathering process of these patterns. I also wish to thank all industrial partners for their willingness to provide the raw data for the pattern mining. These companies include Areva T&D, John Deere, Kone, Metso Automation and Sandvik Mining and Construction. I also wish to thank Nokia Foundation for their scholarship which has aided me in writing these patterns.

References

1. Veli-Pekka Eloranta, Johannes Koskinen, M.L., Reijonen, V.: A pattern language for distributed machine control systems. Technical report, Tampere University of Technology (2010) ISBN 978-952-15-2319-9.
2. Alexander, C.: The Timeless Way of Building. Oxford University Press, New York (1979)
3. Samek, M.: Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems. 2 edn. Newnes, Newton, MA, USA (2008)

Functional Safety System Patterns

Jari Rauhamäki¹, Timo Vepsäläinen¹ and Seppo Kuikka¹

¹Tampere University of Technology, Department of Automation Science and Engineering
P.O. Box 692, FI-33101 Tampere, Finland, {jari.rauhamaki@tut.fi, timo.vepsalainen,
seppo.kuikka}@tut.fi

1 Introduction

Safety is an emerging issue that is constantly gaining more importance in many sectors including industrial control and machinery applications. Safety is required by laws and regulations and demanded by customers. Consequently, vendors are required to offer safety certified products. As compatibility with safety regulations and standards is demanded increasingly, a cost-effective safety system design process gives an edge to a vendor. Design pattern approach can help to simplify the design process and provide an easy to understand view to safety-related systems.

Design patterns are popular in the field of software engineering and plenty of patterns have been published. Contradictorily, in the field of control and safety engineering patterns have not been studied and published in large volumes. Nevertheless, we see pattern as a valuable tool in the domain of control and safety system development. During our studies on safety-related software applications in a machinery domain, we have been able to identify some promising patterns related to safety control applications.

1.1 Safety-Related System

The patterns in this paper focus on safety-related systems. Before we can define a safety-related system, we need to define the term safety in general. IEC 61508 [1] (in the part 4) states that safety is “freedom from unacceptable risk”. This is a generic definition covering all kinds of specific definitions related to e.g. physical, financial or social damage or hazards [2].

A safety-related system can now be defined as a system that “implements the required safety functions necessary to achieve or maintain a safe state for the EUC (Equipment Under Control)” and “is intended to achieve, on its own or with other safety-related systems, the necessary level of safety integrity for the implementation of the required safety functions.” [1]. Thus, safety-related system reduces the risk of an undesired event on acceptable risk level by affecting the operation of a system. A safety-related system may reduce the risk of a hazard by reducing either the probability or the consequences of the hazard, which are the factors of the risk.

1.2 Safety-Related System Development

Typically safety-related systems are developed to be compliant with standards regulating the devices and machines of the considered domain. The safety-related system must take into account the requirements proposed by the standard. For instance, IEC 61508 is a generic standard for the development of safety-related systems whereas EN ISO 13849-1 [3] is focused in safety of machinery applications. The standards define a set of methods and techniques to be used in the development process. In addition, the standards propose requirements on the structure and the operation of the system. IEC 61508, for instance, defines how a safety-related system must be developed. In contrast, domain specific standards are typically more concerned with the safety-functionality of the system. That is, what kind of safety functions the system must implement (emergency power off, a shutdown if people enter working area, etc.).

The amount of requirements and constraints introduced by standards, laws and regulations related to development of safety-related systems is considerable large. Development of such system is somewhat bureaucratic and burdensome process due to various techniques and methods required to be used in together with high level of documentation. The patterns in this paper try to ease the burden considering safety-related system development in context of architecture and generic principles applied in safety systems. The patterns raise and provide solutions to some of the fundamental questions in development for safety system to co-exist with a control system responsible to handle the normal control operations of the system under development.

1.3 Pattern Overview

This paper presents six patterns related to the development of machine and industrial process control applications. The patterns have not been mined from industrial applications developed by companies because the access to documents is restricted. However, according to conversations and interviews with professionals in the industrial control domain, some of the solutions the patterns describe are known in the industry and utilized in the industrial control domain in both machinery and industrial process control. Others are our own ideas of possible patterns.

The patterns do not (yet) belong into larger pattern language. However, it seems that there could be potential to build a pattern language based on these patterns (or include them into such a language) considering safety-related E/E/PE (Electrical/Electronic/Programmable Electronic) systems and their development. The patlets (i.e. short descriptions) of the patterns are provided in **Table 1**.

Table 1. Pattern patlets

Pattern	Patlet
Separated safety	Development of a complete system according to safety regulations is bureaucratic and slow process. Therefore, divide the system into basic control and safety systems and develop only the safety system according to safety regulations.

Productive safety	Control system utilizes advanced and complex corrective functions to keep the controlled process in the operational state. These functions are very hard to implement in safety system. Therefore, implement the corrective functions in basic control system and use simple(st) approach for safety system.
Separated override	Safety system must be able to override basic control system whenever systems control same process quantities. Therefore, provide the safety system with separate actuator to obtain safe state.
De-energized override	Safety system must be able to override basic control system whenever systems control same process quantities. Therefore, let safety system use de-energization of the basic control system's actuator(s) to obtain safe state.
Safety limiter	Safety system must be able to override basic control system whenever systems control same process quantities. Therefore, disengage the basic control system completely from the actuator and let safety system control the actuator. Route the basic control systems output to safety system and let safety system treat the control value so that safe operation is ensured.
Hardwired safety	Development of safety-related application software for simple safety function is bureaucratic, time consuming and costly. Therefore, instead of a software-based solution, use a hardware-based safety system.

Relations of the patterns are presented in **Fig. 1**. The most centric pattern is the SEPARATED SAFETY. The arrows illustrate a typical order of usage of the pattern (solid line arrows). That is, when a pattern is applied the patterns it points at can be considered. However, this is merely a typical approach and one should take too restrictive. The patterns can also be used as standalone solutions. The dashed arrows between patterns indicate similar patterns i.e. patterns that solve the same (or similar) problem but with distinct solutions due to context and forces.

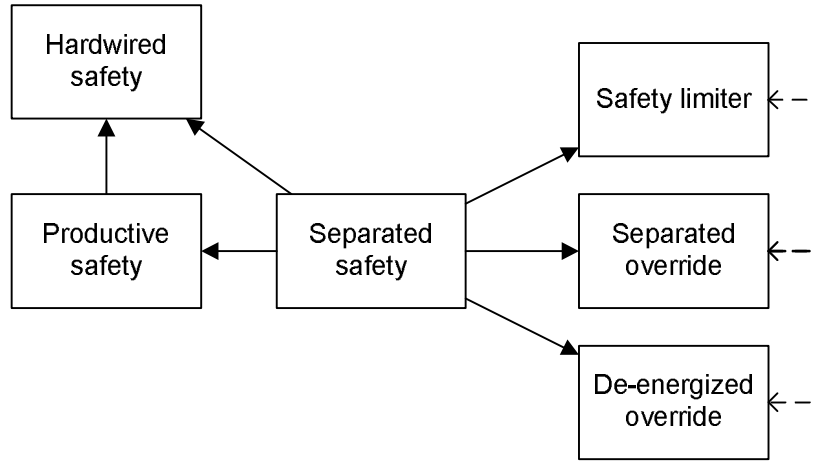
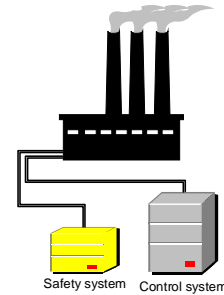


Fig. 1: Relations of the patterns

2 Separated Safety

Context

A control system for a work machine or an industrial process needs to be designed and developed. According to performed hazard and risk analyses, the system to be controlled is capable of causing physical or financial harm to the environment or people working in its surroundings. Because of the possible risks, the functional safety of the system must be ensured with a safety system that must be developed according to appropriate standards and possibly certified by authorities. The safety-related standards restrict development process, tools and methods and in addition require usage of various techniques and measures not directly needed to develop a control system.



Problem

Designing the whole system according to safety standards is costly, bureaucratic and a slow process.

Forces

- **Safety:** The functional safety of the system must be assured with a system compliant to dominant safety regulations.
- **Standards:** Safety-related standards such as IEC 61508 [1] require independence between safety-related and non-safety-related systems.
- **Cost-efficiency:** Development of the whole control system according to safety standards would be difficult and increase the development costs substantially.

- **Cost-efficiency:** Use of certified components in the whole control system would increase the hardware costs substantially.
- **Suitability:** Certified components and processing units with limited instruction sets may not enable development of all required basic control functionalities. For example floating point arithmetic is not supported by all safety certified processing units, which makes it hard to use such units in basic control system development.

Solution

Divide the control functionality into two separated systems: basic control system and safety system. Requirements for the whole control system are first divided into safety-critical requirements and non-safety-critical requirements. Typically, the safety-critical requirements are related to deviation and possibly hazardous situations whereas the non-safety-critical requirements are related to normal operational conditions and the intended use of the system. Safety-critical functionality is then designed and implemented into a safety system according to safety standards. Non-safety-critical functionality is designed and implemented into the basic control system. This frees the development of the basic control system from requirements of the safety system. Thus, all kinds of (non-certified or safety approved) devices, tools, methods, instruction sets, etc. are utilizable in basic control system development.

The safety system and the basic control system are separated from each other so that the correct functioning of the safety system is not dependent of the correct functioning of the basic control system. If necessary, the safety system may utilize certified hardware such as sensors, actuators, buses and safety PLCs. The basic control system may utilize the same components provided that it is not capable of disturbing the correct functioning of the safety system; otherwise, it must use different components. Because the basic control system is separated from the safety system, the requirements of safety standards do not apply to the development of it. Separation also potentially enhances the development process of the system. Because the systems are separated there are no (or very little) dependencies between the systems. Thus the systems can be separately developed in parallel and by different development teams (which is beneficial from diversity point of view).

Fig 2. illustrates a separated safety system within a process. The process box represents a (sub)process under consideration. Basic control system controls the process. The safety systems are insulated from the basic control systems. The safety system as well as the basic control system has their own hardware (controllers, actuators, sensors, etc.) That is both systems affect the same process but they operate independent from each other.

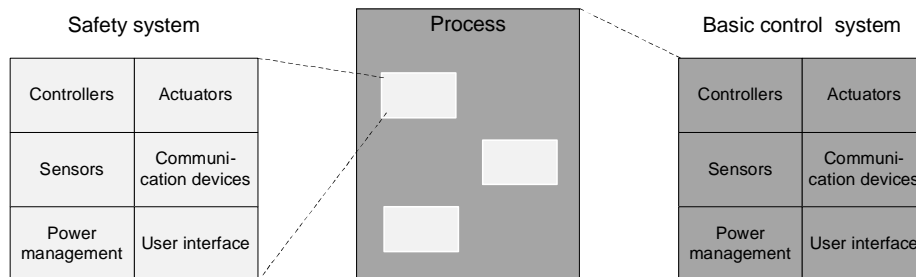


Fig. 2. Illustration of separated safety system within a process

Consequences

- + Safety of the system can be achieved with an appropriate safety system.
- + Basic control system development may utilize the development process, tools and techniques preferred by the company – not the ones required by safety standards.
- + Full instruction set tools, computing units and components can be used with the basic control system.
- + As the safety is ensured with a separated system, the basic control system does not need a certification.
- + The development costs of the basic control system can be reduced. This is due to basic control system doesn't have to be developed according to the requirements considering safety systems.
- + Because the safety and basic control system are separated, the development of them can be outsourced separately or they can be developed independently from each other by different development teams. This can also affect positively to the schedule of the whole project.
- Two separated applications must be developed and they may require different instrumentation.

Resulting Context

The resulting solution consists of two separated systems which can be developed separately so that the independency of the safety system from the basic control system can be proved to the authorities.

Related Patterns

The PRODUCTIVE SAFETY describes how the responsibilities and complexity of the safety system can be decreased even further and how activations of the safety system can be reduced to necessary situations only.

The HARDWIRED SAFETY pattern describes a way to avoid need for safety related application software in safety system development.

The SEPARATED OVERRIDE, DE-ENERGIZED OVERRIDE and SAFETY LIMITER patterns provide solutions to override basic control system with safety system so that safety system has the final word in system's operation.

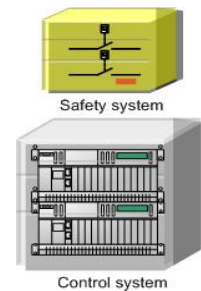
Known Usage

The solution is widely used in Finnish process industry and also known in the domain of mobile work machines.

3 Productive Safety

Context

A control system for a work machine or an industrial process is being developed and the SEPARATED SAFETY pattern has been utilized so that the control system functionality is divided into two separate systems: basic control system and safety system. However, some user requirements are safety-related although not safety-critical. Quite often, these kinds of requirements can be due to customers willing to avoid the financial consequences of the activations of the safety system, such as shutting down the machine or plant. Or, the system should remain operable near (but still within) the safety limits so that the productivity of the system can be increased.



Problem

How to divide the responsibilities between the safety system and the basic control system so that the system would remain operational as long as possible and as near the safety limits as possible?

Forces

- Safety: The safety of the system must be ensured in every foreseeable situation.
- Productivity: operating near safety limits often increases the productivity of a process.
- Economy: Financial impacts of, for example, running down a power plant or a paper machine are dramatic and not desired unless it is not absolutely necessary to achieve safety.
- Recovery from deviations: The customers and users of the system want the system to try recovering from and correction of disturbances. The recovery algorithms may require complex and advanced functionality and/or logic.
- Development process: Development of complex and advanced algorithms and functionalities in context of safety-critical system is considerably burdensome and costly due to restrictions and requirements of safety-critical system development.

Solution

Implement corrective functions in basic control system and use simple(st) approach for safety system. The purpose of the corrective functions is to keep the system within the desired operation range and handle disturbances such as rapid changes in ambient temperature. These functions also implement protective operations such as interlocks to prevent undesired operations.

The corrective functions for disturbances are typically necessary for fulfilling the requirements of clients. The algorithms and techniques to implement corrective functions are potentially complex and advanced and thus they are problematic if implemented in safety system according to safety standards. These functions should be implemented in basic control system. The safety system, on the other hand, can be designed to be as simple as possible. In many (though not all) cases, the system is in the safe state when the system is not powered. So, without trying to recover from disturbances, the safety system can often be designed to take relative simple measures to drive the system into safe state when critical safety limits are violated.

The scope of the basic control system is widened to include functions meant to keep the system in its operation region in which safety system never activates. In the basic control system, the corrective actions (interlockings) can be as complex as required to achieve the goal. **Fig. 3 a)** represent situation before application of the PRODUCTIVE SAFETY. The dashed line represents division of responsibility between the basic control system and the safety system. In **Fig. 3 b)** the responsibilities of basic control system are widened i.e. safety system takes control later.

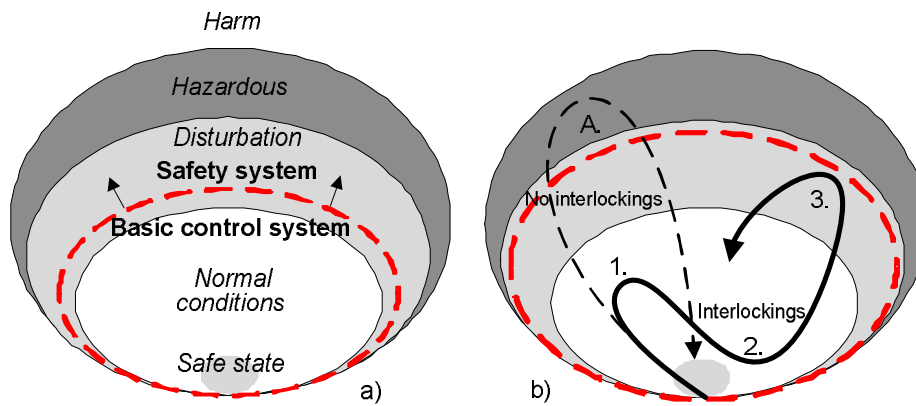


Fig. 3. System states of operation from control system point of view.

In **Fig. 3 b)** a possible operation path of a process is illustrated (solid line arrow). The process is first in normal operation condition. Soon the state starts to shift towards disturbed state. In point 1, the interlockings prevent the process from entering to the disturbed state. If there were no interlocking the state would have changed to hazardous where the safety system had taken control (point A) and returned the system into safe state (dashed line arrow). After point 2 the process state leaves normal conditions and enters the disturbed state. The system state begins to move towards normal conditions from disturbed conditions in point 3 due to successful interlocking operations. The main idea of the interlockings is to prevent the system entering the hazardous state that would cause the safety system intervention and e.g. complete shutdown of the system.

Example

In a chemical process, an example of a corrective action of a basic control system could be relieving pressure with a relief valve before actual safety limits whereas a safety system could be designed to stop all heaters and pumps related to the process.

Consequences

- + The safety of the system can be achieved with a safety system that is designed to be as simple as possible which makes it easier to develop and certify.
- + The corrective actions that are required for operating the system near safety limits can be implemented without the need to follow safety standards. Operating the system near safety limits often increases the productivity of the system.
- + The system is not shut down by the safety system unless absolutely necessary.
 - The complexity of the basic control system is increased

Resulting Context

The safety of the system is still ensured with the use of the safety system that can be developed to be as simple as possible. The scope of the basic control system, on the other hand, is widened to include functions the purpose of which is to keep the system in a safe region so that the safety system gets never activated.

Related Patterns

The SEPARATED SAFETY pattern describes how to divide the system into safety-critical and non-safety-critical parts.

The HARDWIRED SAFETY pattern describes a way to avoid need for safety related application software when implementing simple safety functions.

4 Separated Override

Context

A control system for a work machine or for an industrial process is being developed. The SEPARATED SAFETY pattern has been utilized so that the control system functionality is divided into two separate systems: basic control system and safety system. The separation of safety and basic control systems is followed strictly (to e.g. enable easier certification process). The separated systems may in some places control the same functionalities or process variables.



Problem

Safety system and basic control system control the same process quantities. Consequently, the basic control system may interfere with operation of the safety system, but the safety system must have the final word on system operation.

Forces

- Safety: Safety control system must always be able to drive the system into safe state (i.e. a state in which system minimizes the risk of damaging itself or people around it) regardless of the state of the basic control system
- Non-interference: Separation of the basic and safety system is the main concern
- Hardware: Additional hardware is not a problem in terms of cost, space, and weight etc.
- Hardware: Actuators with sufficient safety level and suitable functionality can be hard to find (e.g. hydraulic proportional control valves with SIL 3 certificate are not too widely available)

Solution

Use separated actuators for safety and basic control systems. The purpose of safety actuator is to drive the controlled process variable into safe state as controlled by the safety system. The basic control system operates its own actuator for control purposes and the safety system operates a separated actuator for safety purposes.

The principle of the architecture is presented in **Fig 4**. A separate actuator for safety system is added in the points in which the safety system needs to have control over the basic control system. The safety control system must always be able to override the control system's operations. Ensure that the safety function cannot be circumvented or bypassed in any way by the control system. The safety system has an ability to drive the state of the controlled quantity to the safe state regardless of the control system state. Only the actuator controlled by the safety system has to comply with safety (standard) requirements whereas the basic control system's actuator can be chosen freely.

Notice, that safety function typically wants to either fully enable or disable the controlled variable. The actuator controlled by the safety system is then placed in parallel or in series in terms of the control actuator respectively. The safety actuator must be placed so that the normal control system cannot bypass it. Consider also the order of the actuators. The safety actuator may be positioned before or after the basic control actuator depending on the system.

Example

A possible application of the separated override principle can be found for example in processes in which steam flow to a heat exchanger is controlled (see **Fig. 5**). The

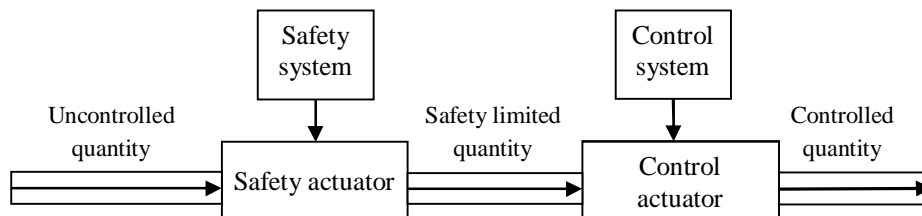


Fig 4. Principle of separated override

safety function is to prevent overheating of the heated element. The control system is responsible for controlling the flow using a proportional valve. In addition the steam line is equipped with a safety valve controlled by the safety system. Now, regardless of the basic control system the safety control system may halt steam flow in the heat exchanger. Notice that the figure illustrates only a part of a system.

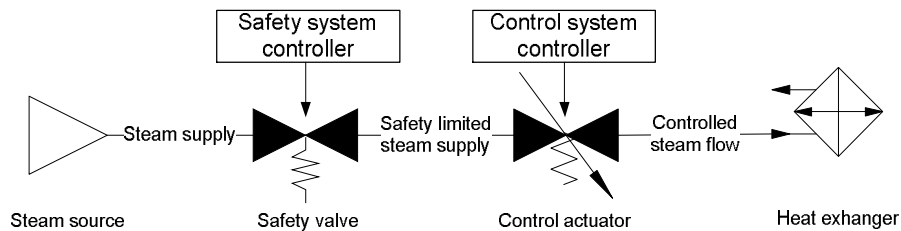


Fig. 5. Example of separated override in steam flow control

Consequences

- + (Complete) separation between safety and basic control systems is obtainable
- + Safety is retained by safety system if control system fails (e.g. control actuator gets stuck)
- + Selection of the safety system actuator is disengaged from selection of the basic control actuator
- + Main control actuator(s) can be chosen freely without need for certain safety properties
- + Simplest possible safety system approach can be chosen in terms of actuator type (binary on/off actuator is typically sufficient)
- + The approach doesn't restrict selection of data transfer method from controllers to actuators (e.g. safety system may use analog signaling and the basic control system Flexray bus etc.)
- Increased cost due to additional safety actuator
- Increased weight and space requirements due to additional safety actuator
- Additional hardware may increase the complexity of the hardware system

Resulting Context

The safety control system may override the control system in all situations and thus safety is not dependent on the basic control system. An actuator for the safety system is added to the system in each point the safety system needs to have control over the basic control system.

Related Patterns

The MERGED SAFETY ACTUATION describes how total amount of safety actuation hardware and cost can be decreased by merging actuation of multiple safety functions into single actuator.

The DE-ENERGIZED OVERRIDE PATTERN describes an alternative solution to similar problem. The pattern describes how separated actuator for safety system can be

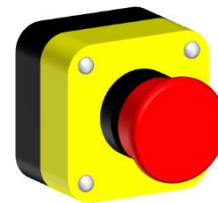
replaced with a relay (or similar switching device) to reduce need for actuator hardware.

The SAFETY LIMITER describes an alternative solution to similar problem. Safe operation can be achieved by circumventing the control signal calculated by the basic control system through a safety function that limits the value without a need for an additional relay or other hardware. Consequently no additional hardware is required besides the control actuator, but potentially complex safety-related application software needs to be developed.

5 De-energized Override

Context

A safety function for a control application is being designed. The SEPARATED SAFETY pattern has been utilized so that the control system functionality is divided into two separate systems: basic control system and safety system. The system has, in context of the considered safety function, a well-defined safe state. The safe state is always the same regardless of the state of the system, but tripping of the safety function can depend on various aspects.



Problem

Safety system and basic control system control the same process quantities. Consequently, the basic control system may interfere with operation of the safety system, but the safety system must have the final word on system operation.

Forces

- Safety: Safety control system must always be able to drive the system into safe state (i.e. state in which system minimizes the risk of damaging itself or people around it) regardless of the state of the basic control system.
- Safety: The safe state is always the same.
- Separation: Separation of the safety and basic control system is required.
- Hardware: Additional hardware is problematic in the system (e.g. in terms of space and weight).
- Cost-efficiency: Additional separate actuator for safety system increases cost of the safety system.
- Hardware: Actuators with sufficient safety properties and suitability to safety and control tasks are available.

Solution

Use de-energization of the basic control system actuator(s) to obtain safe state and to override the control signal of the basic control system. The safety function must have a well-defined safe state that is independent from the system state. When the safety function is tripped, the control actuator is de-energized and the actuator takes a safe

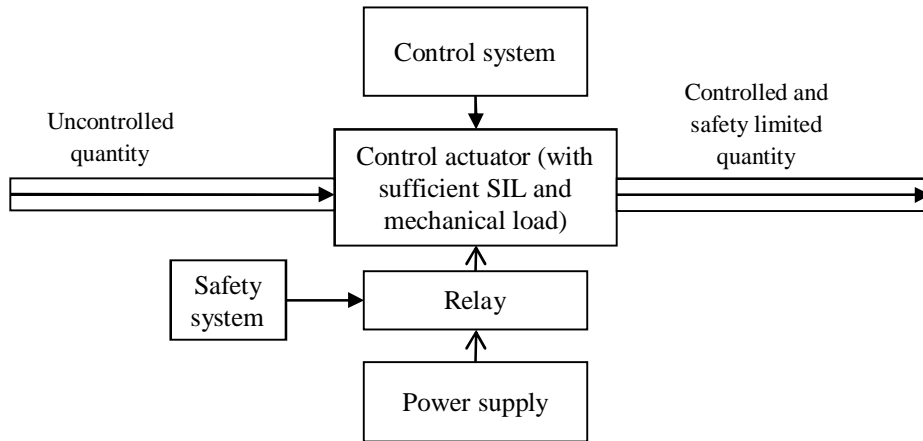


Fig. 6. Safety actuation through de-energization

state. Now, regardless of the basic control system input the actuator is in safe state and the basic control system is overridden by the safety system.

In this approach a dedicated safety actuator is replaced with a relay (or similar switching device) that is used to de-energize the control actuator. The safety system controls the power supply of the control actuator through the relay. The principle of the approach is depicted in **Fig. 6**. The control system is responsible to provide the normal control signal to the actuator. Whenever the safety function trips, it de-energizes the actuator, which then enters the predefined safe state forced by a mechanical load.

A relay is typically more compact in terms of size and weight than a dedicated safety actuator (a valve for example). However, the control actuator has to be mechanically loaded (e.g. spring loaded) to be able to enable the safe state when de-energized. This increases the size and weight of the control actuator. In addition the control actuator (or at least the mechanical loading system) has to be compatible with the dedicated safety integrity level of the safety function. This may increase the cost of the actuator significantly as complex actuators with high safety integrity properties are not cheap. Notice that also the relay controlled by the safety system needs to have sufficient safety properties as it is part of the safety function.

The architecture is not suitable for all cases. The architecture may not be applicable if there is a risk that the actuator type used in the control is not able to obtain a desired state to enable the safe state. For example, the architecture must not be used if there is a risk of blocking of the actuator (e.g. in some hydraulic systems/environments due to impurities).

Ensure that the safety function cannot be circumvented or bypassed in any way by the control system. If such architecture is used, it must be ensured that the safety system is able to drive all the actuators effecting the application of the considered safety function into safe state.

Example

A possible application of the de-energized override principle can be found for example in processes in which the flow of steam to heat exchanger is controlled (see **Fig 7**). The safety function is to prevent overheating of the heated element. The control system is responsible for controlling the flow using a proportional valve. The safety functionality is actuated with the same valve through de-energization. The safety system controls a relay through which the power to the control valve is supplied. When safety system trips the power is cut from the valve actuator and spring loading turns the valve into the safe state position. Now, regardless of the basic control system the safety control system may halt steam flow in the pipe. Notice that the figure illustrates only a part of a system.

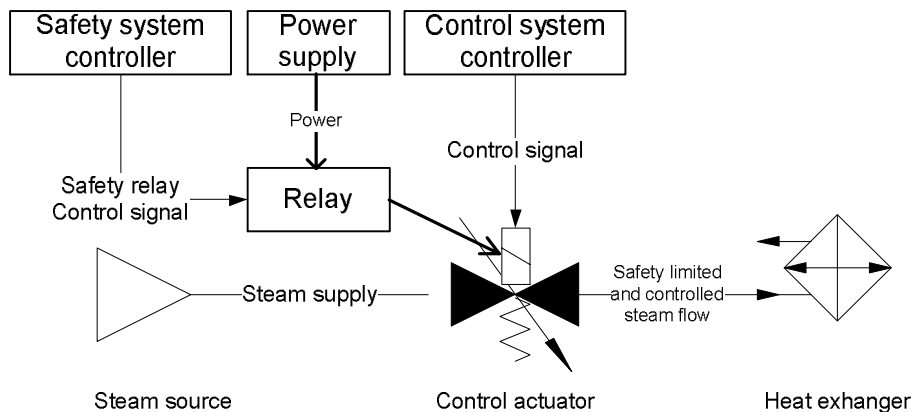


Fig 7. Example of de-energized override in steam flow control

Consequences

- + Safety is retained by safety system regardless of the basic control system
- + Separation between the safety system and the basic control systems remains
- + No need for additional safety actuator → reduced weight, size and (potentially) power consumption
- + System wide power loss results a safe state of the actuator
- Only one safe state can be applied
- Requires an actuator suitable to the control task with sufficient safety properties...
- ...that is mechanically loaded to actuate safe state when de-energized...
- ...which typically results an expensive actuator
- Still requires a relay or similar switching device to detach actuator from its power supply

Resulting Context

The safety control system may override the control system in all situations and thus safety is not dependent on the control system.

Related Patterns

The SAFETY LIMITER pattern describes an alternative solution to the problem. Safe operation can be achieved by circumventing the control signal through a safety function that limits the value without a need for an additional relay or other hardware. Consequently no additional hardware is required besides the control actuator, but potentially complex safety-related application software needs to be developed.

The SEPARATED OVERRIDE pattern describes an alternative solution to the problem. Safety system is provided with dedicated actuator. This gives safety system more alternatives to actuate the safety function, but also requires additional actuator.

The SAFE WHEN UNPOWERED pattern describes a generic process design related principle of constructing a system so that safe state is obtained whenever power is lost.

6 Safety Limiter

Context

Safety function architecture is being designed. The SEPARATED SAFETY pattern has been utilized so that the control system functionality is divided into two separate systems: basic control system and safety system. However, the systems may share information. Amount of hardware needs to be minimized and thus no dedicated actuator or other supporting hardware for the safety system can be added. To retain safety in all situations the safety system has to have full control over the process variable(s) affecting safety.



Problem

Safety system and basic control system control the same process quantities. Consequently, the basic control system may interfere with operation of the safety system, but the safety system must have the final word on system operation.

Forces

- Safety: Safety control system must always be able to drive the system into safe state (i.e. state in which system minimizes the risk of damaging itself or people around it) regardless of the state of the basic control system
- Safety: Safety function is very context specific and may require arbitrary controls
- System independence: Separation between basic and safety control systems can be compromised
- Hardware: Additional hardware is problematic (e.g. in terms of space and weight)
- Software: Software has no drawback of hardware in terms of spatial requirements and can be duplicated for free in mass produced devices
- Cost-efficiency: Additional separate actuator for safety system increases cost

- Hardware: Actuators with sufficient safety properties and suitability to safety and control tasks are available

Solution

Disengage the basic control system completely from the control/safety actuator. Calculate the control signal in basic control system and limit the value with safety system. Provide the control value calculated by the basic control system to safety system that checks the safety conditions and limits the control value to safe range if necessary. The safety control system controls the actuator.

An illustration of the architecture is provided in **Fig 8**. The basic control system calculates the control value for the actuator. The basic control system may use any kind of algorithm freely from restrictions of safety regulations according to which the safety system has to be developed. The basic control system then passes the control value to the safety system. The purpose of the safety system is to limit the control value so that safety is retained. The safety system is developed to conform with safety regulations so it shouldn't contain any unnecessary functions. The safety system alone is connected to the actuator.

The basic control system's controller and the safety systems controller can either be separated devices or a single device running both safety-critical and non-safety-critical functions. The latter approach requires additional measures to be taken (see Related Patterns). The actuator needs to have sufficient capabilities to satisfy the safety standards' requirements and it must also be suitable for the control task. Thus the actuator may prove expensive.

If the SEPARATED SAFETY pattern has been applied, no additional actuation hardware is allowed and use of the DE-ENERGIZED OVERRIDE pattern cannot be applied due to operation of the safety function (no single safe state position can be defined), the solution is the only considerable approach. Another possibility would be development of whole control algorithm according to safety requirements, but this results a costly development process and violates the SEPARATED SAFETY pattern.

Data communication from basic control system to safety control system is always a potential risk from safety control system's point of view. However, the systems may communicate if certain conditions are met. Consequently the software architecture of safety control system needs special attention. These conditions are not too restrictive as in general communication through global data or similar data structure is not

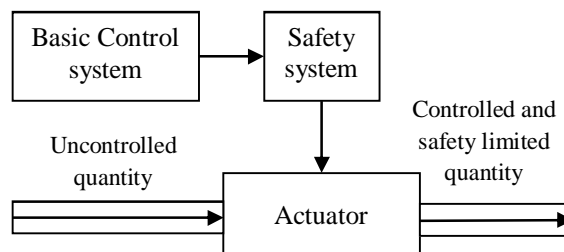


Fig 8. Principle of software safety limiter based basic control system override

recommended. Communication should be established through a well-defined interface e.g. a class method or message bus message (with strict structure).

Example

A potential application for the pattern is usage in situation in which operator controls movement of a machine (or part of it). Let's consider a modern combat aircraft. A pilot controls the (instable) aircraft by indicating the direction she wants the plain to steer. The basic control system then calculates how the winglets need to positioned to steer the aircraft into desired direction. However, the pilot can only withstand limited amount of g-forces so the aircraft protects pilot by restricting the aircraft movement with additional safety system¹.

The architecture of the system can be as follows. The pilot steering inputs are first processed by the basic control system, which uses advanced algorithms to calculate optimum control values for the winglets (actuators). This includes at least the basic stability considerations and some other functionality such as ramping of the values. As the calculations complete the value is passed to the G-force protection safety function. The G-force protection then calculates the amount of force observed by the pilot. If the protection function notices that too great strain is directed into the pilot it can restrict the control value. The architecture of the system is illustrated in **Fig 9**.

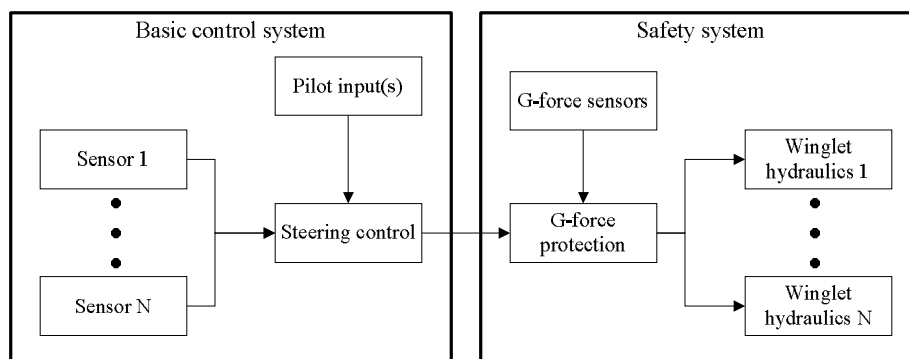


Fig 9. Combat aircraft steering control with safety limiting functionality

Consequences

- + Complex basic control algorithm can be used without need to develop them according to safety standards
- + Advanced safety functions can be established through software based safety functions. Certain type of safety functions are very hard to implement with hardware solutions such as restricting motion into complex shaped space.

¹ Ok, the software of a combat aircraft is probably safety-critical all the way (or at least developed as one), but for the sake of this example we consider these control tasks as separate functions. In practice the stability of the plane is probably prioritized over G-force protection, but we don't cling into this little drawback. The functionality could be, however, used during training for instance.

- + No need for dedicated safety actuator
- + The solution is cost-effective especially when system is mass produced as software is cheaper to multiply than hardware
- Breaks separation between safety and basic control systems, but if implemented rationally, strong (though not complete) independence can still be achieved
- The safety system may increase the delay in control compared to situation without additional safety controller between the basic controller and the actuator
- Data communication from basic control system to safety control system inflicts a potential risk...
- ... and thus software architecture of the safety safety-critical system and the data communication in general requires special care due to data obtained from system that doesn't conform with safety standards

Resulting Context

The basic control system is disengaged from the actuator. The safety controller has full control over the actuator and can override the basic control system's control signal. A dependency between basic control and safety system is created and basic principle of the SEPARATED SAFETY pattern is violated.

Related Patterns

The DE-ENERGIZED OVERRIDE pattern describes an alternative solution to similar problem. The pattern describes how safety and basic control system can use shared actuator without need use data communication between safety and basic control system controls. However, number of possible safety operations is restricted to one, i.e. only one safe state can be obtained.

The SEPARATED OVERRIDE pattern describes an alternative solution to similar problem. Safety system is provided with dedicated actuator. This gives safety system more alternatives to actuate the safety function and provides complete separation between basic control and safety systems. However, the approach requires additional actuator.

The SHARED CONTROLLER pattern describes how dedicated safety and basic controllers can be merged into one device and thus decrease the total device number even further.

The CASCADED SAFETY CALCULATION pattern describes what kind of software architecture can be used to manage input stream from basic controller while doing versatile safety considerations and limitations affecting single actuation unit according to the basic controller's initial control values.

7 Hardwired Safety

Context

Implementation for relatively simple safety functionality is being considered. For instance, the PRODUCTIVE SAFETY pattern has been applied so no advanced



functionalities are implemented in the safety system. The safety system cost is not hardware orientated, i.e. the final product is not going to be mass produced.

Problem

Development of safety-critical application software is costly and provides no real benefit in context of the considered safety function.

Forces

- Cost-efficiency: Development of safety-critical software is costly and time consuming and unnecessary safety certified controllers are expensive
- Simplicity: Safety systems should be simple and understandable
- Maintainability: Safety systems should be easy to maintain
- System design: Commercially-off-the-shelf (COTS) safety capable hardware products for various environments and purposes are available.

Solution

Instead of a software-based solution, use a hardware-based safety system. Hardwired safety systems can be used to implement simple and generic safety functions such as over and under temperature, pressure and speed related to a process variable. Certified COTS hardware for such generic functions is available. Advanced and custom safety functions are, however, easier to implement with custom software-based safety applications. As the safety system for the considered safety function includes no safety-critical software, maintenance of the system becomes easier. There are no problems between different software versions.

Remove the need for safety-controller and safety-critical application software² by establishing direct link between sensor(s) and actuator(s). Simple hardware based logic components can be used if logic is required between device connections. For instance an actuator trigs if both sensors agree so (i.e. the sensors are connected with AND-gate). Do not use controller based logic if it requires software development.

The sensor(s) measures system state and trigs the actuator(s) to apply the safe state when defined conditions apply. The devices need to be compatible in terms of communication. That is, the sensor must provide suitable output signal and the actuator needs to be able to use the signal generated by the sensor. Any communication method from analog signal to message bus (as far as compatible with safety regulations) is applicable.

The following guidelines can be used to identify safety functions, which could be implemented with hardwired solutions.

- Firstly, the safety function requires no complex logic or calculations. In principle, any logic can be implemented with simple hardware devices (such as logic gates), but in practice merely simple logic (AND, OR, NAND, etc. or small scale combinations of them) functions are sensible as hardware implementations.

² In this context safety related custom application software refers to software that is developed for the system under design. Embedded software (e.g. firmware) in safety certified COTS devices is not counted as (custom safety related) application software.

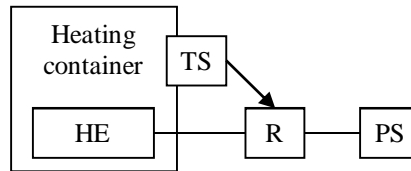


Fig. 10. Safety function with hardware implementation

- Secondly, there should be a well-defined trigger for the safety function. In this context a trigger means an event in the system that trigs the safety function active (e.g. liquid level rises above a maximum value). Simple logics can be used to connect several trigger conditions (e.g. liquid level high and exhaust valve closed). However, advanced conditions are problematic, e.g. mean values.
- Thirdly, the safety function should be able to actuate the safety function in relatively simple manners. That is, positioning multiple outputs to arbitrary states is problematic whereas controlling single binary output is considerably easier.

Example

A possible utilization target for hardwired safety function would be, for example, the over temperature protection function of a heating container. In the requirements for the safety of a plant, a requirement for an over temperature limiting of the main heating container of the plant is given. As the plant development is (at least nearly) a unique project, hardwired safety systems are used to minimize safety related application software. The solution is depicted in **Fig. 10**.

A temperature sensor (TS) is applied to the container being monitored for over temperature. A relay (R) is applied between power source (PS) and heater element (HE) of the container. The sensor output is connected to relay input and when the temperature reaches a predefined value, the output of the sensor goes off and relay opens. The power source is detached from the heating element and temperature of the container no more rises. A safe state is obtained.

Consequences

- + No need for safety-critical application software
- + Burdensome and expensive development process of safety-critical software is omitted
- + No need for dedicated safety-related controller → may reduce number of devices of the safety system
- + Intelligibility of the system is increased
- Advanced features of the safety system are hard to implement with pure hardware solutions
- Expansion and further development of hardwired solutions are harder than software based solutions

Resulting Context

Safety function is implemented with no safety-critical application software, using hardwired safety system. Safety system controller is not needed (in context of the

considered safety function) and thus one device can be potentially left out from the system (unless needed for other safety functions).

Related Patterns

The operability of the system can be improved utilizing the PRODUCTIVE SAFETY pattern. Advanced safety functionalities can be implemented using software.

8 Future Work

The presented patterns provide a possible starting point for a pattern language considering safety-related system development. Our objective is to continue working towards such language and we already have some new patterns under development. However, the domain of safety-critical system development is vast considering standards, practices and other regulations. Consequently we are forced to focus on some section of the domain. Possible focus areas within safety-related systems are for example work machine domain and software architectures.

9 Acknowledgments

The authors would like to thank Farah Lakhani for shepherding the paper. In addition our thanks belong to Veli-Pekka Eloranta, Ville Reijonen, Dirk Schnelle and Joonas Salo for their valuable feedback in the VikingPLoP 2012.

References

1. IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission (2010)
2. Wikipedia: Safety, <http://en.wikipedia.org/wiki/Safety>
3. EN ISO 13849-1. Safety of machinery, Safety-related parts of control systems, Part 1: General principles for design. International Organization for Standardization. 2006.

Patterns Related to Software Updating for Machine Control Systems

Ville Reijonen
{firstname.lastname}@tut.fi

Department of Software Systems
Tampere University of Technology
Finland

1 Introduction

In this paper we will present two software updating related patterns for machine control systems. A machine control system is a software system that is specifically designed to control a certain hardware system. This special hardware in turn operates some work machine, which can be a forest harvester, a drilling machine, elevator system etc. or some process automation system. Some of the key attributes of such software systems are the close relation to the hardware, real time requirements, safety issues, fault tolerance, high availability, and long life cycle of tens of years. In most large machines distribution plays a major part in the control systems as the different parts of the machine are physically far from each other and they must communicate with each other in order to perform their functionalities.

When the system has life cycle of tens of years, something is bound to change or break during that time. Hardware components available in the future might be different than now; old software might not be able to use the hardware of even function on the hardware. It might be hard even to access all components physically. There might be new accessories available in the future which could not be fathomed by that time the original software was designed. Even the software might have new functionalities and bug fixes. There is multitude of reasons why one would like to update their system.

The patterns in this paper are part of a larger collection which was collected during years 2008-2011 in collaboration with industrial companies. Some real products by these companies were inspected during architectural evaluations and whenever a pattern idea was met, the initial pattern drafts were written down. These draft patterns were then reviewed by industrial experts, who had design experience from such systems. After these additional insights, the current patterns were written. The published patterns are a part of a larger body of literature, which is not yet publicly available. All these patterns together form a pattern language, which consists of more than 70 patterns at the moment.

2 Patterns

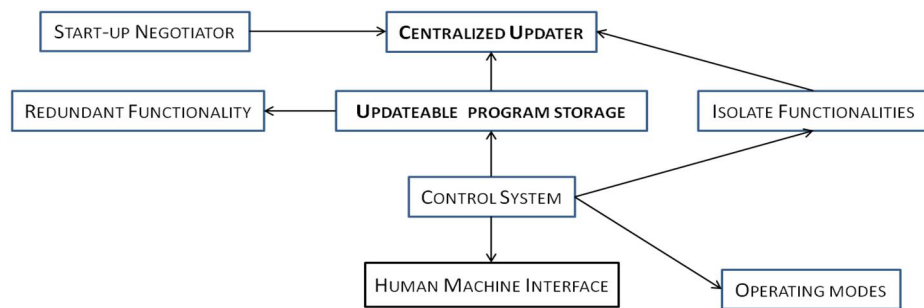
Other patterns are referenced in this paper using SMALL CAPS. Patlets of the patterns that are referenced or are in this paper are presented in Table 1. Pattern names written in bold are presented in this paper.

Table 1. Patlets of patterns presented and referenced in this paper.

Pattern name	Description
CENTRALIZED UPDATER	Update of entire system should be convenient for the operator. Therefore deliver compatible software together bundled in a single package. Create a centralized updater component which distributes the software from the bundle to nodes when necessary and aims for consistent system state.
CONTROL SYSTEM	How to implement a work machine that offers interoperability between systems and is highly operable with good performance? Implement control system software that controls the machine and can communicate with other machines and systems.
HUMAN MACHINE INTERFACE	How to provide information on the state of the machine and way to modify the state of a machine? Provide the the operator a screen and pointing devices so that she may interact with the software.
ISOLATE FUNCTIONALITIES	What is a reasonable way to create an embedded control system for a large machine? Distribute the system into subsystems according to their functionalities. Interconnect these subsystems with the bus. Use multiple interconnections between subsystems if necessary.
OPERATING MODES	How to make sure that only the functionalities which are required can be used in current operating context? Design system that consists of multiple functional modes. These modes correspond to certain operating contexts. The mode only allows usage of those operations that are sensible for its operating context.
REDUNDANT FUNCTIONALITY	How to ensure availability of a functionality even if the unit providing it breaks down or crashes? Clone the unit controlling a critical functionality. One of the units is active and other is in a hot standby mode. If the controlling unit fails, hot standby mode unit takes over controlling the functionality.
START-UP NEGOTIATOR	How to verify that required devices for base configuration are available and find out which features

	can be enabled based on the other found devices? Design a central node that gathers information on available devices and what they provide and require. After deadline check the information.
UPDATEABLE PROGRAM STORAGE	Software should be changeable in a system with long life cycle. Therefore for each node place the software on rewritable persistent storage which is also large enough for future needs. It should be possible to update the software over the bus. Any update failure should not prevent from trying again.

The relationships of the patterns listed above in Table 1 are drawn to the Picture 1. Pattern names written in bold are presented in this paper.



Picture 2. A graph representing the relationships between patterns.

2.1 Updateable Program Storage

... you have CONTROL SYSTEM which has software. A shipped system might be in production and in service for tens of years, production systems sometimes located in hard to access locations. Due to long service time the original requirements rarely cover the needs of the future. There are often needs to update and modify the system during its lifetime.

* * *

Software should be changeable in a system with long life cycle.

While the time passes, software may evolve by receiving new features and bug fixes. System setup might go through changes that move it out of the capabilities of the original software. This is almost a certainty in systems with long planned life cycles up to thirty years. Thus, a system who's parts cannot be updated will literally be stuck to past. If renewal is not an option, the systems value will diminish greatly by every new requirement which it cannot fulfill.

Usually cost is a big issue when system is being built. This usually results in decisions which lock the system design constraints to known needs. One way of saving costs is to put the software on read only memory (ROM), but then to update the software the ROM has to be changed. Other way to save is to limit the size of the memory so that it fits just the current version of the software. This might cause problems if in future the software is larger than one made previously.

In working machines one would prefer to cover well all easily damageable components such as electronics. To be manually changeable the chip would need to be accessible. It might be embedded to such a location that it is hard to reach or detach. Attaching separate cable for flashing or replacing the chip might require extensive amount of work and expertise, which is not always available. Changing the software should be easy and relatively fast.

When a system is updated it involves a risk of failure. This risk should be minimized but as it is not always possible, there should be ways to mitigate aftereffects. It should be possible to retry any operation and even reverse it when required. It would be even preferable if this could be tried without any external help as such is not always available or even anywhere close to the location where the system resides.

It is difficult to predict all possible future needs for software as those needs have not yet materialized. The only way to enable the flexibility needed by the future is to be able to change the software to meet the new requirements. These requirements might lead, for example, to faster operations, better accuracy, better energy efficiency, different hardware used etc. None of these can be met unless the software is changeable.



Therefore: **Place the software on rewritable persistent storage which is also large enough for future needs. It should be possible to update the software over existing wire. Any update failure should not prevent from trying again.**

* * *

The usual way for the update to commence is to set the system to separate update OPERATING MODE. The program which does the updating could be either reside on separate ROM space, be part of the program residing on the persistent storage or be loaded to the memory as first step of a update process over the wire. Depending on how much memory the system has the update can be either transmitted as whole or fed in suitable sized blocks. When the update commences an under-update flag is set. The update program rewrites the persistent memory with the data it has received. When all the data has been transmitted, the update is verified, under-update flag is removed and the system is rebooted.

When the update is commencing, the program code under update should not be used as it would probably lead to undefined behavior and unknown errors. It should not be possible to run the update by mistake and the updating functionality should be shielded from unintentional or malicious use. Updating software over the wire can be simple what comes to the update operation but it requires additional safeguards. The node should be shielded with access rights or different usage mode should be required for the update functionality.

There should be a way to flag the system as being under update. The system should not be used when under update. There might be partition in the memory or separate memory area reserved for flags. Only after successful update the system should be flagged again as active. This way the system would not be used even by mistake if the update fails. The update may fail or hardware might break during the update. Therefore, after writing process, the result or functionality should be verified to know that the update was successful. The verification can be done, for example, by counting checksum from the written result and comparing it to expected result or doing byte by byte comparison. Only if the verification is successful under-update flag may be removed. If a system boots and under-update flag is set, the software should try to enter update OPERATING MODE. This is one way to start update process that ensures that no program code is in use. This can also be handy method to recover and start over, for example, in case of power failure.

Updating the updater program is always risky and should be done only if it is really necessary as it will render the system useless in case of failure. If the updating pro-

gram resides on separate ROM space it cannot be updated, but the system can always boot to updating mode. If the program resides on persistent storage the update program may also start, but only if previous update has not overwritten the update program with garbage. If the program is loaded over the bus by a small stub handling the loading during boot, the failed update may be recoverable if the stub is not corrupted. If the program is loaded over the wire while changing to update mode, it might impossible to recover if the update has failed as the program might not be able to commence so far to start the loading again. Therefore it is highly advisable to have to update program a part of the boot-up routines.

Update can always fail due to power loss, bad connection, faulty hardware, etc. In any case a system which is in transitory state should not be used and under-update flag guarantees that the system does not become active. If there is a risk that disturbed update might render the system unusable, there could be unrecoverable memory errors or the system has been classified critical, it might be good idea to apply REDUNDANT FUNCTIONALITY pattern and duplicate the memory slots so that there is always one complete working copy of update program or contents of persistent storage available. Continuous boot cycles should be detected and the system put to disabled state as last resort if the device does not booting continuously.

There should be more storage available than what is required by the first version of the software as new versions usually gains new features which consume additional space. How much more space is needed depends on the purpose of the system and its development path. If systems are shipped out to the world in large quantities it might be reasonable to plan future requirements to find suitable storage size. Consequently, if only few systems are shipped, making such a plan probably costs more than what can be saved by using smaller storage. If a system does not have enough storage for the envisioned functionalities, squeezing the software to smaller size will make programming much more complicated and costly.

When there system structure is distributed by ISOLATE FUNCTIONALITIES pattern, it is often the case that more than one subsystem should be updated. Software version discrepancies between different subsystems might be a problem as the subsystems need to work together. With CENTRALIZED UPDATES a system consisting of subsystems can updated to consistent state.

* * *

Updateability raises the value of the system over its lifetime. When the system can be updated, new features, bug fixes and other enhancements may be received. Some of these might be covered by service contract while others might be add-ons to existing system. By providing easy way of updating, less is wasted on non-value adding operations to get the update available. Still, an easy mean to service software should not be seen as a permission to use the end users as software testers.

Update can always fail. It might fail continuously rendering the device unusable. It might be that the memory can't handle rewrite. In any case a device might be lost due to the act of updating and that is a risk always worth considering. More difficult it is to replace faulty part, the more safeguard there should for recovery. This naturally raises the complexity of software compared to one on non-updateable system.

Update activity should not be accessible unintentionally or mistake by some noise on the bus. It should be hard to activate it maliciously. Suitable safeguards should be put in place. Designing and implementing these safeguard will add costs.

Rewritable persistent storage in suitable quantities adds the necessary flexibility needed to support needs of the future. This adds design costs but in a machine which consists of multitude of nodes which are all updateable, the same design can be used all over. When larger run of machines is produced the cost per node should be not be an issue. Still, rewritable storage is not as cheap as ROM is, but gained flexibility should be more than adequate to offset the costs. Cost for replacing buggy ROMs cannot even be compared to price of running software update.

* * *

In harvester's boom a grappler module is updated to new software version. The machine is put to stop. The service person connects USB-stick containing update for the grappler to cabin pc USB-port. The updating tool sends the new software to the grappler module's updater program over the bus block by block. After writing the code is read to verify that its cyclic redundancy checksum (CRC) is correct. If the update was successful the machine can be operated with functional grappler module.

2.2 Centralized Updater

... you have a distributed CONTROL SYSTEM where each node can be updated over the bus with UPDATEABLE PROGRAM STORAGE. The bus has reasonable throughput so that it is sensible to update nodes over the network. START-UP NEGOTIATION is in place and provides information what features are available based on existing nodes. HUMAN MACHINE INTERFACE is in place so that the operator is able to interact with the software. Still updating each node one by one is tedious manual task. It is also potentially error prone as the operator may forget to update a node or nodes.

* * *

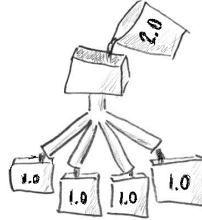
Update of entire system should be convenient for the operator.

The nodes in the system communicate with each other to function as a whole. Therefore, each node should have a software version which is designed to work together with others. If nodes have mixed untested set of software, there might be unknown compatibility issues. Certain versions of software are developed and tested together and there is better knowledge of their compatibility than for others.

If new node is added to the system, it might have different version of software than the rest of the system. It should be possible to update the node or the whole system to a compatible version level so that support for new software would be guaranteed. It would be even better if the updating would be doable on side of installation or regular maintenance.

In case of any failures, it should be possible to automatically retry updating. If updating does not work the operator should be informed on the situation and offered alternative possibilities for solving the problem if possible. It should not be the task of operator to try to figure out what went wrong and how to proceed.

Updating system node by node manually is time consuming and cumbersome. As a humane lapse may cause problems as described previously, such a task should not be left to be handled by a person. It would be preferred if the software update delivery would be automated and the operator could decide on activation.



Therefore: **Deliver compatible software together bundled in a single package. Create a centralized updater component which distributes the software from the bundle to nodes when necessary and aims for consistent system state.**

* * *

To be able to decide what is the current system state and what nodes require updates START-UP NEGOTIATOR is enhanced so that it gathers information on software versions from each node during start-up. Enhanced START-UP NEGOTIATOR will function as central clearinghouse for version information gathering. This version list is reported to centralized updater, which in turn will check if nodes have software from same software bundle.

If the system versions are not consistent or if never update bundle is available, operator is informed via HUMAN MACHINE INTERFACE that an update should commence or that new bundle is available. If the operator agrees to update, the centralized updater distributes the updates to the nodes requiring update one by one. Each of these nodes updates itself using the mechanism provided by UPDATEABLE PROGRAM MEMORY pattern. A sensible order of updating would be doing in the order of dependencies. This would guarantee that at least basic services are compatible even if more advanced are not, if there is some major problem with the update process later on during the process. Naturally during the update the system should not be operational.

If the update of a node fails, update will be retried using the retry mechanism provided by UPDATEABLE PROGRAM MEMORY. If, after a couple of retries, the update cannot be completed successfully, it has to be determined if the node is functional at all. If the node is functional there are three recourses to the situation: get a new node which can be updated, disable the node or try to use other software bundle for the whole system. There might be multitude of hardware versions and just some do not work with all the software. The operator should be notified on the failure and told about alternative ways of trying to solve the situation.

If a node is detected during start-up to have a software version which is not from the same software bundle as the rest of the system, it should be updated with software from the software bundle to make it compatible. Software version disparity could be either caused by a node which has been replaced, or by a new node which is added to

the system. This requires the centralized updater to contain software for every known device which can appear in to the system.

If new hardware is added, it is noticed during the start-up by START-UP NEGOTIATOR. Before the new hardware can be used, the operator is queried if the added node should be set-up or disabled. It might be possible that there is no support for the new device in the currently used software bundle. In this case, the whole system has to be updated to a newer version of the software bundle where support is available before hardware may be activated.

Sometimes there is a new software bundle available for the whole system. When the operator loads the new software bundle from a media to centralized updater, the bundle is verified as valid before it is stored. After the bundle has been stored to centralized updater the whole system can be updated from a single point with the contents of the bundle. There can be multiple bundles available at the same time, so that any software bundle can be used if necessary.

When there are multiple bundles available on the centralized updater, the operator can choose which bundles should be used for the machine. This is for the convenience of the operator as it might be inconvenient time to do a whole system update or the operator might prefer the way the system functions with the older software bundle.

To combine compatible and tested software together, all the software for the system and potentially used devices are combined into a single software bundle. Basically software bundle is a compatibility tested snapshot. It has all the software from certain date for all hardware which the system may have. Still, it should be noted that as the software bundle has software for a wide range of hardware, it is possible that not all combinations are tested and therefore there might be setups which do not work together.

* * *

The whole update process is doable from a single point. If newer hardware is available but there is no support for it in older software bundle, the hardware cannot be taken into use unless the software is updated for the whole system. With this limitation older software bundle can always be used instead of newer if, for example, the newer software has some issues or the operator prefers the older software.

After an update the whole system software should be compatible. Of course having software which makes this all happen adds complexity to the system and creates higher potential of failure. Still, it is hard to predict what would be setup of a system beforehand. Especially it is hard to know what new devices will available in future.

Creating software bundles is hard work. It involves a lot of compatibility testing, but it will be impossible to test all combinations which could exist in wild. When you

the long life cycle and changing hardware components to the mix, it will be impossible to test all hardware and software versions.

Malicious use and self created software bundles are always a possibility which should be thought out, if so it seems to be necessary.

* * *

A drilling machine is stopped. An operator insert an USB stick to the cabin computers USB port. The system notices the inserted media, selects the software bundle and verifies that as a valid. After this the bundle is extracted to the centralized updater. The system notifies the service person of the new available system update and queries if it should be installed. After the service person confirms, the system commences with the update. The software is sent from the centralized updater to each node in the system one by one over a bandwidth limited CAN bus. After the last device is updated and success is verified, the machine is restarted with updated software.

3 Acknowledgements

I want to thank my colleagues Johannes Koskinen, Marko Leppänen and Veli-Pekka Eloranta for their help. In addition, I want to thank all industrial partners for their valuable cooperation in our pattern mining process: Metso Automation, Kone, Sandvik Mining and Construction, John Deere, Areva T&D. As last but not least, I thank for my better half Ewa, who let me have these writing sessions required, when free time is limited, to write this prose down instead of doing diaper changing duties.

Doing Small-scale Agile Projects Efficiently and Profitably

Ilkka Laukkanen <ilkka.laukkanen@futurice.com>

Futurice Oy

1 Introduction

Futurice is a software agency of about 140 people with offices in Helsinki and Tampere, Berlin and London. We offer consulting, design, development and training services to clients from diverse industries.

Our projects are generally done on a tight schedule and without complete, concrete specifications up front, instead evolving quickly from a concept to a polished product. In terms of project outcomes and customer satisfaction we have been very successful in working in this agile manner. The aim of this paper is to document some of the methods we've used to be agile and to keep projects on time and in budget while helping our customers with their business, and keeping both them and our employees happy while doing it. These patterns that are key parts in our way of working and contribute greatly to being successful and profitable while working on small-scale projects in an agile way. The patterns are

- END TO END,
- PERIODIC DEMO, and
- NO HANDOFFS.

In the context of this discussion, efficiency is taken as a measure of the time during which employees assigned to a project are actually able to do work on deliverables, in proportion to the billable hours they spend on that project. Efficiency is decreased when developers or designers have to hunt for information, write exhaustive documentation instead of new features—unless documentation itself is the aim—or generally do anything that isn't directly advancing the project towards its stated goal, and is instead waste [5]. Some inefficiency is inevitable, and some is necessary: sticky notes have to be moved on Kanban boards, project email has to be read and meetings attended and so on, and this is well and good, but increasing efficiency would allow to shorten project timespans, which would make for more compelling quotations.

Profitability is defined in the short term as avoiding overruns—in the case of black box projects—and in the long term as being able to charge realistic rates despite heavy competition. Overruns are caused by many things, but generally are due to either solving the wrong problem, or solving the right problem but in the wrong way.

Small-scale here means both brief and of limited scope. Agility is on one hand an umbrella term for the methods we use, but it is also a desirable element of the customer's perception of us.

2 END TO END

2.1 Context

In our projects customers often present only a preliminary concept, which we then start to develop and design further. They look for us to help develop the idea into a concrete, useful product or service and see this plan through to completion. This requires a sense of ownership of the product, and pride in the work that we do. We want customers to have the sense that their idea is in good hands and that we care about what comes out of it. The more we show we care, the more trust is placed in us.

From the customers' viewpoint, getting everything from consulting and design through implementation and training to life-cycle management from one place is beneficial, because it avoids expensive knowledge transfer phases between subcontractors. When projects are short, the extra effort this requires toward comprehensive documentation starts to become a prohibitively large fraction of the total project workload.

Internally there needs to be cohesion between design and development throughout the project. For a given total project budget B , an overrun is guaranteed if a designer spends $\frac{1}{2}B$ coming up with a design that will cost $\frac{3}{4}B$ to implement.

For an agency such as ours, building repeat business and keeping customer satisfaction high is important, because a lot of business depends on referrals, recommendations and reputation. When we take an interest in the customer's business and actively help them develop it, we make an investment towards our own future. To achieve this it is necessary not just to cultivate a capable team, but also an environment where communication towards the customer is open.

2.2 Problem

How can we foster customers' trust in us and build a foundation for repeat business in a field where much is uncertain in the outset?

2.3 Forces

- Customer trust is important to future business
- Repeat business is important to the bottom line
- Project deliverables are often defined in terms of preliminary concepts that should be developed further before implementing them
- Projects are so short that collaborating with third parties takes up a disproportionate amount of time
- Project budgets have to be reasonably split between design and implementation

2.4 Solution

When pitching and planning, aim to deliver a complete solution; get input from designers, developers and people with experience in maintaining systems; come up with the next chapter in the product's story and pitch that as well. During the project get developers and designers working closely together. Use PERIODIC DEMO so everyone knows what is being built and shares the vision.

2.5 Rationale

When everybody is involved from the start, problems can be identified earlier: technical experts with domain knowledge can identify problems in the design, and life-cycle management specialists can point out issues with the selected implementation technologies, for example. Corrective measures can be taken early on when their impact is still minor.

2.6 Related Patterns

This is related to CROSS-FUNCTIONAL TEAMS [1] in that tight co-operation between different parts of the organization is needed to keep everybody on track, and VISION [4] in that a shared vision is required, although it is more a joint effort than a creation of the product owner.

3 PERIODIC DEMO

3.1 Context

When a product is taken from a seed idea to a complete implementation, a lot of communication is needed to ensure that the thing that is being built matches with the customer's perception and need. Communicating via non-functional prototypes or sketches is fraught with the danger of misunderstandings, omissions and other sources of errors. There is no better way of showing progress, getting feedback and building a common understanding than handing the customer a working product as soon as possible, and keeping on handing them when the product improves and develops.

3.2 Problem

How can we manage customer expectations and correct misunderstandings and omissions in project requirements with minimal impact on project duration?

3.3 Forces

- It is very difficult to gauge if specifications truly reflect customer desires when working from a rough concept
- It is equally difficult to effectively communicate project progress with numbers alone
- Errors should be found and fixed as soon as possible before they become any more costly

3.4 Solution

Commit to having a working first version of the product ready for the customer to use after the first iteration, and a new working version at the end of every subsequent iteration.

3.5 Rationale

The customer's unspoken and undocumented preferences will quickly become apparent as they get to use even a rudimentary demo version of the product. It will also help determine future direction, as not all ideas that sound good in meetings and on paper end up translating well in actual use. Furthermore, since the basis for the changes we end up making is the working product, we have confidence that we're using the best, most accurate information available, instead of modifying unrealised plans whose validity we cannot fully know.

Also, by striving to keep the product in a state where it could conceivably be released at any time, we force ourselves to do integration at an early stage. Integration is a major source of technical risk for any project, and doing it early is an effective way to manage it, as Reinertsen writes in "Managing the Design Factory", pages 224–229 [7].

Finally, the later corrections are made, the more rework they cause, which in turn causes project overruns, and they have a direct negative impact on profitability.

3.6 Related Patterns

This pattern is similar to BUILD PROTOTYPES [2] in that work is being done to better understand the requirements, especially latent ones, but these products are not thrown away, instead becoming product increments [3]. In small projects we can refactor very aggressively to keep the product from becoming prototype spaghetti.

4 NO HANDOFFS

4.1 Context

The communication overhead discussed in END TO END applies not just between many subcontractors collaborating on a project, but naturally also between teams and individuals within one company. If a customer comes to a representative with their ideas and problems, and that representative talks to a team—and in extreme cases maybe that team then talks either directly or via a proxy to e.g. an overseas team—the process becomes a game of Chinese whispers. This often resembles the situation described in PERIODIC DEMO, where consensus cannot be reached due to poor communication, and quality suffers as a consequence.

4.2 Problem

How can we make sure that communication between stakeholders is as effective as possible, so that no time is wasted waiting for information or even building the wrong thing?

4.3 Forces

- The project team inevitably needs to communicate with the customer to get clarifications and other input
- The greater the number of steps in a communications chain, the larger the chance of information becoming distorted, and the longer that information takes to traverse the chain
- Managing personal communications does take up time that could be otherwise spent in project work

4.4 Solution

Remove impediments to direct communication and foster team engagement with the customer by:

- having regular face-to-face meetings;
- providing a feedback channel (such as a bug tracker) for the customer;
- having the customer provide contacts for direct technical questions.

4.5 Rationale

When the team has direct access to the customer, and conversely the customer to the team, feedback and guidance can be given unimpeded and all the important questions can be asked directly from the people that have the answers. This saves time and lessens chances of miscommunications caused by extraneous mediators.

This requires a somewhat active customer, but our experience is that they are often willing to get engaged in these less rigidly defined projects, being invested in them and excited about them to begin with.

Intuitively it would seem that this approach results in much of the developers' time being spent handling feedback, but this is not necessarily the case. In "Leading Lean Software Development" Mary and Tom Poppendieck cite two examples of removing handoffs resulting in major benefits for the customer (pp. 20–21, 222–223) [6]. Instead of the team being swamped with feedback, the mental models of the team and the customer were more aligned, and as a result the customer was more satisfied with the end product.

The same goes internally too. The less rigid the product definition, the more communication is necessary (see "Managing the Design Factory", pp. 113–115) [7]. This means that team members must be in contact at all times when working with the kind of vague specifications we usually have.

References

1. Cross-functional teams. ScrumPLoP published patterns. Website, referenced Feb 9 2012. <https://sites.google.com/a/scrumplp.org/published-patterns/team-pattern-language/cross-functional-team>.

2. James O. Coplien and Neil B. Harrison. Organisational patterns: Build prototypes. Wiki, referenced Feb 2 2012. <http://orgpatterns.wikispaces.com/BuildPrototypes>.
3. Lachlan Heasman. Regular product increment. ScrumPLoP published patterns. Website, referenced Mar 5 2012. <https://sites.google.com/a/scrumplop.org/published-patterns/value-stream-pattern-language/regular-product-increment>.
4. Lachlan Heasman. Vision. ScrumPLoP published patterns. Website, referenced Feb 9 2012. <https://sites.google.com/a/scrumplop.org/published-patterns/value-stream-pattern-language/vision>.
5. Mary Poppendieck and Tom Poppendieck. *Lean Software Development*. Addison-Wesley, 2003.
6. Mary Poppendieck and Tom Poppendieck. *Leading Lean Software Development*. Addison-Wesley, 2010.
7. Donald G. Reinertsen. *Managing the Design Factory*. The Free Press, 1997.

Self-configurator

Pietu Pohjalainen

Department of Computer Science
University of Helsinki, Finland
Pietu.Pohjalainen@cs.helsinki.fi

Abstract. Traditional object-oriented design patterns often focus to improve flexibility of software. A usual pattern in traditional design patterns is to introduce a new layer of dynamic abstraction that gives an opportunity to parameterize behavior at runtime. A trade-off for this flexibility is often increased complexity and reduced performance. We present a pattern, the *self-configurator*, for resolving symptoms of unnecessary indirections introduced by many standard object-oriented patterns. It allows the use of dynamic structures without paying the associated runtime cost.

Keywords: Software maintenance, automated software engineering, self-configuration.

Intent

A famous quote states, “*All problems in computer science can be solved by another level of indirection*”. This statement is a common driver for many traditional object-oriented design patterns. The less often cited continuation to the phrase states: “*but that usually will create another problem*” [1]. The self-configurator pattern’s intent is to document and automatize the rules for resolving internal dependencies that are introduced by the use of other patterns.

Motivation

Software is full of hidden dependencies, where a seemingly innocent change can cause malfunction or crashing. Programmers have learned to defend themselves and their colleagues and customers from excessive rework by designing their software in a way that is resilient to future changes. As an illustrating example, we will use a dynamic data structure for implementing a simple processor for the Command design pattern [2].

Let's consider the code in Figure 1. It first registers three objects for handling different commands, and then repeatedly reads in a command and dispatches following arguments to the given command.

```
class CommandProcessor {
    static Map<String, Cmd> funcs =
        new HashMap<String, Cmd>() {{
            put("print", new PrintCmd());
            put("noop", new NoopCmd());
            put("quit", new QuitCmd());
        }};

    private static Scanner scanner =
        new Scanner(System.in);

    public static void main(String a[]) {
        while(true) {
            String cmd = scanner.next("\\w+");
            String args = scanner.nextLine();
            funcs.get(cmd).Execute(args);
        }
    }
}
```

Figure 1: Code for a command line processor

For our discussion, the interesting property in this code lies in how the processor uses a dynamic data structure as the storage for the registered commands. Using a dynamic structure makes it easy to add new commands at later time. In contrast to implementing the same functionality by using e.g. a switch-case construct and hard coding the possible commands into the structure of the command processor, this dynamic solution makes the program easier to modify.

This flexibility is gained with the minor runtime cost of using a dynamically allocated data structure with every command fetching being routed through the object's hashing function. Although the runtime cost is small, it still adds some memory and runtime overhead, since the generic hashing implementation cannot be optimized for this specific use case. For example, the standard Java implementation for `HashMap` allocates the default value of 16 entries for the map implementation. In this case, only three of the entries are used, as

shown Table 1. Also, when fetching the command object for a given command, a generic hashing function is used, which also gives room for optimization.

0	<i>null</i>
1	<i>null</i>
2	<i>null</i>
3	<i>null</i>
4	<i>null</i>
5	<i>null</i>
6	<i>NoopOb</i>
7	<i>null</i>
8	<i>null</i>
9	<i>null</i>
10	<i>null</i>
11	<i>null</i>
12	<i>null</i>
13	<i>QuitOb</i>
14	<i>PrintOb</i>
15	<i>null</i>

Table 1:
HashMap default layout

With this discussion, we can see characteristics of accidental maintainability in our example. With accidental maintainability we mean that in this case the solution uses a dynamic data structure for handling a case that does not require a dynamic solution. Namely, the set of available commands is a property that is bound at design time, but a structure that allows runtime binding is used. There are a number of reasons for implementing the command processor in this way. The map implementation is available in the standard class library, its use is well known and understood among programmers and for many cases, and the induced overhead is negligible. Yet another reason can be the lack of viable

alternatives in current pattern knowledge. In cases where any overhead should be minimized, introducing this dynamic structure purely due to implementer's comfort would not be good use of scarce resources.

Solution

An alternative solution for this example is to create a specific implementation of the map interface that is statically populated to contain all the required elements. This would make it possible to use context-specific knowledge of the structure in implementing the command fetching system: instead of using a fully generic hashing table, more memory and runtime efficient, specific hash table and hashing functions for these three commands could be implemented.

The self-configurator pattern resolves this problem by introducing a configurator component to this structure.

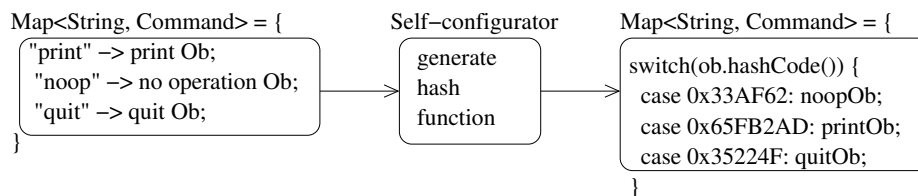


Figure 2: Self-configuring function for the command processor

Figure 2 illustrates the configuration process. The self-configuring component reads the static list of commands and generates a specific hashing function for this set of commands to be used. Now the runtime and memory overhead of generic hashing is avoided. The hash generation function is bound (i.e. executed) at the same time as all other parts are compiled. This way, the runtime overhead can be minimized. However, the design-time allocation of command names and associated functions still enjoys the flexibility of defining the command mapping as a well-understood, standard Map interface.

There is a degree of freedom in placing this generative part in the binding time continuum. The hash generating function and associated hash map generation can happen as part of the normal compilation process, or it can be delayed up until first use of the command processor object. As usual, earlier binding time gives opportunities for optimizing for that special case, while delaying binding gives more flexibility and possibilities to use contextual information to determine behavior.

Applicability

There are many situations when you can apply this pattern. First of all, the pattern is applicable when you are using dynamic structures to guard against changes that a future developer might be performing. In the example in the previous section, the dynamic mapping structure gives defines a clear place for implementing additional commands.

However, this flexibility is gained by introducing additional runtime cost.

Another scenario where you can find this pattern useful is if you need to provide characteristics of one code site to parameterize another routine. An example of this case can be e.g. a dependency between a set of different algorithms performing a computation upon data that is held in the database. Each algorithm requests certain set of data, but you want to separate the database fetching code from the algorithm's processing code. In this case, you can introduce a self-configuring component to analyze each specific algorithm and to automatically produce optimized queries for each algorithm without introducing a dependency between the query site and the algorithm.

Optionally, the pattern can also work as an external reificator by exposing the details of the processed dependency via a dependency interface. This allows programmatic access to characteristics of this dependency. In the previous section's example, this kind of dependency lies between the statically allocated list of commands and the command-line processing loop.

Example structure

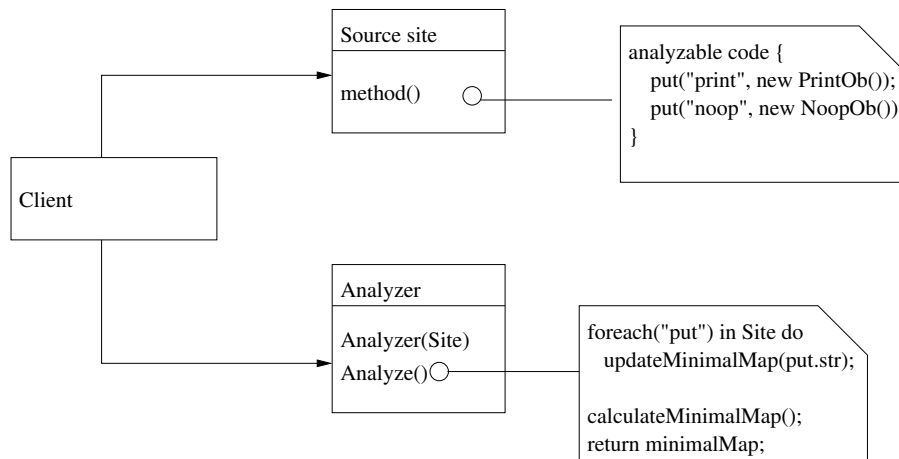


Figure 3: Example structure for the pattern

Figure 3 shows an example structure of the use of the pattern. In the Analyzer part, the self-configurator analyzes se source site and its

relevant properties. When asked to analyze its parametrized code site, it constructs the minimal version of the site and returns it to the client. The client can now use the minimal version instead of the original.

Participants

In general, the pattern deals about computationally resolvable dependencies between code artifacts. The resolver component for configuring the dependencies can be implemented in various ways, depending on the contextual needs. Typical variations for the resolving process can be e.g.:

- Compilation time configuration
- Instantiation time configuration
- Runtime configuration

Usually, the later this configuration is done, the more information is available for the configurator. However, earlier resolving usually offers opportunities for better performance and more options for further optimization.

One scenario for the participants to collaborate in runtime configuration is as follows:

Client instantiates the analyzer, with a parameter that defines the source site to be analyzed

Analyzer reads in the source site definition, and resolves the wanted properties of the source.

Collaboration

The results of the self-configuration can be characterized as intrinsic or extrinsic. In intrinsic mode, the pattern implementation represents a substitute for the analyzed dependency site; e.g. the implementation for the command processor would represent itself as a map from Strings to Commands.

In extrinsic mode the self-configurator analyzes a dependency site and drives another object's configuration based on the results.

Drawbacks

When using the self-organizer pattern, the main problem lies in pre-emphasizing the future needs. The self-configurator ought to be able to adjust its structure to meet the needs of future enhancements. Unfortunately, those people who have been gifted with clairvoyance ability do not end up being software developers. So, for the main bulk of architectural work is done based on best guesses.

For example, the self-configurator component implemented in [8] can automatically reorganize database queries based on the database-accessing algorithm as long as the component can analyze the accessing algorithm. If the implementation changes from the structure expected by the self-configurator, then it obviously fails at its job.

Another problem is that writing self-aware code is non-trivial. In many cases, it seems to be outside the scope of project personnel. Although we have argued that in the past we have been able to implement self-organizing components in agile projects with strict time-boxing limits, it might be the case that this property is not generalizable over all software engineering organizations. In many places, even the term metaprogramming might be an unheard term. In these kinds of organizations, it can be better to start improvements by employing the more classical, well-matured productivity improving techniques.

Implementation

In order to analyze a code site for configuring its dependents, there needs to be a way to access the source data. When using compilation-time configuration, all the source code is available for analysis. For instantiation time and runtime configurations the analysis interface is defined by the execution environment characteristics: some environments, such as the LISP language expose the full structure of the program for further analysis; but many current environments do not. Popular alternatives range from byte-code analysis, such as the BCEL library [3] in the Java environment, to standardized API access to program definition, as implemented in .NET's Expression trees, available in C# since its third version [4].

Regardless of the used access method, the configurator component analyzes the dependent's source. Based on this analysis, the dependent is configured to adhere to the form that is required by the source site. In the running example, a possible configuration could be a generation of a minimal perfect hashing table for the different registered commands.

Often the required target configuration varies from one context to another. What is common in different variations is the built-in ability for the architecture to adapt to changes between architectural elements, which help both in maintenance and in gaining understanding of the overall system.

Known uses

To illustrate the idea of self-configuring software components, we present example cases from our previous work and from the industry.

a) Interpreters and compilers

Tim Barners-Lee is quoted of saying: "*Any good software engineer will tell you that a compiler and an interpreter are interchangeable*". The idea behind this quote is that since the interpreter executes code in the interpreted language, it necessarily has the required knowledge for producing the equivalent lower level code. Also the other way applies: the compilation routines for a given language can also be harnessed to build an equivalent interpreter.

This interchanging process can be seen as the self-configuration component. This has been applied e.g. to build compilers for embedded domain-specific languages [5] and to produce portable execution environments for legacy binaries [6].

The self-configurator in this case can build a compiler from an interpreter by analyzing each opcode definition of the interpreter and by emitting each opcode's corresponding code as the code generation step.

b) Self-configuring database queries

Many useful information systems can be characterized as typical database applications: they read data from a database to the main memory, perform an algorithm on the data, and then write the result back into the database.

These types of applications have a dependency between the data that is read from the database, and the algorithm performing the calculations. Within the object-oriented style of programming, an additional object layer is built on top of a typical relational database, creating an additional problem of object/relational mismatch. An approach of building object-to-relational mapping frameworks, such as Hibernate [7] proved to be popular as a bridge between object-oriented application code and relational persistence structures. In order to provide a fluent programming experience for the object-oriented design, *transparent persistence* is one of the key phrases. The promise of transparent persistence means that objects can be programmed as objects, without thinking the underlying relational database.

One of the tools for achieving transparent persistence is the usage of the proxy design pattern [2] to hide if an object's internal state is stored in the database, or whether it is already loaded to the main memory.

However, in many cases this delayed fetching hides symptoms of bad design: the program relies on the slow, runtime safety net implemented with the proxy. A better design would be to explicitly define, which object should be fetched. If the objects to be processed within certain algorithm can be known beforehand, the usage of the proxy pattern can be classified as a design fault.

We have documented the usage of the self-configurational database queries as a tool to improve runtime properties of this case at [8]. In this design, a code analyzer reads in the byte-code of given algorithm and deducts the required queries for prefetching the needed data from the database. This design helps maintenance properties: should the algorithm change for some reason, the fetching code is automatically updated to reflect the change. Another benefit is that on architectural level, the number of database-accessing components is reduced, since this one component can configure itself for multiple cases.

c) Self-configuring user interface components

Component-based software engineering is widely employed in the area of user interface composition. User interface widgets can be developed as stand-alone components, and a new application's interface can be built by composing from a palette of these ready-made components. Pioneered in Visual Basic, the approach has been adapted to numerous architectures.

One of the drawbacks in component-based user interface composing is the need for duplicated binding expressions when programmatically defining multiple properties of user interface components. For example, when defining whether a user interface component is active or not, a corresponding tooltip should be placed. Without sufficient support for cross-referencing to other binding expressions, providing this kind of conceptual coherence in the user interface requires cloning of the behavior defining expressions.

We built a prototype for analyzing these binding expressions in the standard Java environment for building web interfaces, the Java Server Faces [9]. By exposing the structure of the binding expressions to backend code, we were able to reduce the amount of cloned binding expressions by a factor of 3 in a demo application [10].

d) Generating languages for domain-specific queries

The previous examples work in the expression-level abstraction. The approach of self-configuring components can be scaled to component-level. For example, the QueryDSL framework [11] generates internal domain-specific languages in the spirit of fluent interfaces and interface chaining.

For the problem of querying data in a domain model, the QueryDSL framework generates a class structure that reflects the domain model, augmented with a set of querying functions. These querying functions can be used to formulate aggregation, filtering and sorting queries in the standard Java environment. The generative nature of the framework is exploited to build type-safe queries, which is in contrast to the previous model of using generic objects to bring the domain model concepts to the program's structure.

The difference between the QueryDSL approach and the previous examples is the applicability scale. While the previous example work on intra-component level, there is no reason why the approach could not be scaled to component and systems level.

Related patterns

Many traditional object-oriented design patterns [2] can be analyzed and optimized via this pattern.

The self-configurator pattern can be seen as a formalized variation of maintenance patterns [12]. In maintenance patterns, the idea is to document the required tasks to perform feature adding maintenance tasks. In the self-configurator pattern, these tasks are documented in executable code (reconfiguration rules), so that the software can adapt itself to the new situation.

The pattern uses the idea of introspection and reflection from the CLOS meta-object protocol [13] to build the maintenance instructions.

References

1. Diomidis Spinellis. [Another level of indirection](#). In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 17, pages 279–291. O'Reilly and Associates, Sebastopol, CA, 2007.
2. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
3. Markus Dahm, Byte Code Engineering with the BCEL API. Technical Report B-17-98. Freie Universität Berlin, 2001.
4. *C# Language Specification*, version 3.0. Microsoft Corporation, 2007.
5. Conal Elliott, Sigbjørn Finne and Oege De Moor, Compiling Embedded Languages. *Journal of Functional Programming*, Volume 13 Issue 3, May 2003
6. Alexander Yermolovich, Andreas Gal and Michael Franz. Portable execution of legacy binaries on the Java virtual machine, PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, ACM: New York, NY, 2008; pp. 63–72.
7. Christian Bauer and Gavin King. *Hibernate in Action*. Manning Publications, 2004.
8. Pietu Pohjalainen and Juha Taina. Self-configuring object-to-relational mapping queries, PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, ACM: New York, NY, 2008; pp. 53–59.
9. JSR 252: *JavaServer Faces 1.2*. Technical report, Sun Microsystems, 2006.
10. Pietu Pohjalainen. Self-configuring user interface components. In A. Dix, T. Hussein, S. Lukosch, J. Ziegler (Eds.), *Proceedings of the First Workshop on Semantic Models for Adaptive Interactive Systems*, 2010.
11. Timo Westkämper, Samppa Saarela, Vesa Marttila and Lassi Immonen. [QueryDSL reference documentation](#) [online].
12. Imed Hammouda and Maarit Harsu. Documenting Maintenance Tasks Using Maintenance Patterns. In proceedings of CSMR 2004, IEEE Computer Society, pp. 37–47, 2004.
13. Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow, *The Art of the Metaobject Protocol*. MIT Press, 1991.

Pattern definition of MapReduce

Joonas Salo

Department of Software Systems
Tampere University of Technology
PL 553, 33101 Tampere, Finland
`salo.joonas@gmail.com`

1 Introduction

MapReduce is a programming model that was introduced in 2003 by Google [1]. The idea is to split the input data so that it can be processed in small items or groups of items in parallel. At Google the model was first used to build the search index and later it has also been found useful in many other contexts.

This paper was motivated by the success of MapReduce in different environments. There was interest to analyze the reasons for applying the pattern and also differentiating it from similar approaches. This paper first describes the MapReduce pattern and then three different use cases. Each use case applies the pattern in different environments. The two first cases, Hadoop and CouchDB, are production ready open-source solutions for efficient data processing, while the last one, MashReduce, is a more experimental research application for building web mashups.

2 MapReduce pattern

2.1 Context

The amount of data is increasing each year. Data formats vary from text to multimedia files such as pictures and sounds. In other words, the data can exist in any arbitrary format. This data may be part of the application logic, such as user databases, or it may have been collected as a side product of the processes, such as web server logs. The data is not presentable as such, but needs to be processed in some way to extract meaningful information out of it.

2.2 Problem

There is no time to wait for too long for the processing of the data.

2.3 Forces

- Ad-hoc solutions in software world work in small programs, but require much maintenance and stress the developers if the programs grow.
- For processing huge amounts of data a delay of days or weeks can be completely acceptable. For small amounts of data the timeframe can be counted in seconds. In any case, there is not enough patience or money to wait for too long.
- Data has a tendency to grow, so it good to choose a scalable solution.
- Performance can be enhanced in the SW by optimizing the algorithms, and then in the HW by upgrading components such as cpu and memory. Unfortunately, these measures are not always enough, or they can not be taken, so some different approach needs to be taken to reach the desired level of performance.
- There are existing solutions to choose from for big data processing.

2.4 Solution

MapReduce is a programming model that works by splitting the data into small items and groups of items that can be processed in isolation, and thus, in parallel. MapReduce is best described in two levels.

First there is the programming model that defines how the data flows through the map and reduce phases to the output. This idea takes form in the development of the map and reduce functions. They can be very simple, but require some thinking when combined together.

Secondly there is the framework that runs the map and reduce functions. Different frameworks have different characteristics that make them useful for either big data processing in batches, or realtime processing for smaller amounts of data.

The map and reduce functions are the paramount of MapReduce, but the underlying framework is equally important and has surely taken more effort to develop. It does a lot of work shuffling the data around between different processes and different computers, so that the map and reduce functions do not have to worry about the details of distribution. Each MapReduce framework manages the map and reduce functions in different ways. Some of the different details are explained later in the case studies, but in the following is given a general pattern description.

The first thing to understand is, that the data is always split into items. Items are constituted of a key and a value. In the beginning, before running any functions, the split is done automatically by the framework. For example, if there are many text documents, the split may be such that each document is one item where the name of the document is the key and the content of the document is the value.

The splitted data is then processed either by a map, or a reduce function. The difference is in the granularity, a map function processes each item, while a reduce function processes each group of items. The groups are discriminated

by keys. The reduce operation is heavier, as the framework needs to gather all items with the same key together.

Each map and reduce step filters or transforms the data and changes somehow. The data split is also changed at each step and at some intermediate steps there may be many times more data than at the beginning or at the end of the task. The framework takes care of all this data management in a distributed environment.

2.5 Positive Consequences

- Complex data processing can be split into multiple steps
- The framework does a lot of work in the background so that it is easy to develop the business logic in map and reduce functions
- Map and reduce functions may be reused for different views
- Data processing time can theoretically be divided by the number of nodes and nodes can be added and removed as needed

2.6 Negative Consequences

- The framework is a dependency and it may not be realistic to switch to another framework.

2.7 Related Patterns

A simple choice for distributed computing is to use a queue. The queue is filled with tasks that worker computers fetch whenever they have capacity. This kind of solution may be faster to setup and can accommodate any kind of programs. On the other hand as the data and processing programs are loosely coupled, a lot of automated data handling that a MapReduce framework can provide is not available.

2.8 Resulting Context

Big amounts of the data can be viewed in reasonable amounts of time. But if the input data keeps growing, the processing cluster needs to scale also to cope with that. On the other hand, it might be possible to think about what data really is important and if some of it could be discarded.

2.9 Example

The flow of data from the input documents and through map and reduce to the output is explained in the following wordcount example:

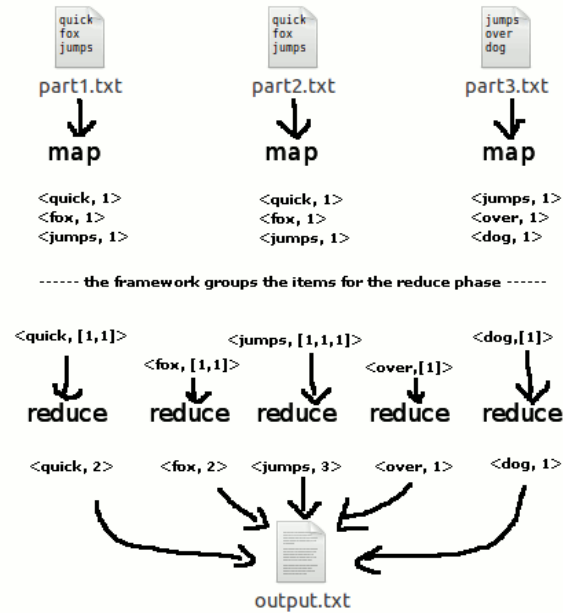


Fig. 1. Wordcount example

In this example, the data is constituted of three text documents that each contain a few words. The data is split in three, naturally by the number of documents, and a map function is run for each split. The map function creates an item for each word it encounters in the document. The key of the item is the word, and the value is 1, meaning that it occurred once. The occurrences of each word could also be counted together at this point, but this is omitted for simplicity. When all maps have run, the framework sorts the items by key for the reduce functions. Then, one reduce function is started for each set of items with the same key. The reduce function simply sums up the values (1+1+..) to produce the final count of that word in all the documents.

```
// Example functions for the wordcount example
map( filename, words ):
    for word in words.split():
        emit( word, 1 )

reduce( word, counts ):
    wordcount = 0
    for c in counts:
        wordcount += c
    emit(word, wordcount)
```

Note that in this, and most other documents, map and reduce may be referred to as functions, programs, agents, or simply as mappers and reducers. usually these terms simply mean the same thing from slightly different point of views, but sometimes there is a distinction that a mapper or reducer means the computer that does the mapping or reducing.

3 Crunching Big Data with Hadoop MapReduce

Hadoop MapReduce is a mature and advanced MapReduce framework. It is the closest open-source equivalent to Googles proprietary MapReduce framework for dealing with really big data.

In Hadoop, the map and reduce functions are run in pairs, first a map and then a reduce function. Either one can also be omitted or replaced with an identity function. To accomplish more complex tasks, the workflow of running multiple mapreduce steps needs to be managed outside of Hadoop. Basically the output of the first task is used as the input for the second task, and so on. There exists many tools managing task chaining automatically.

For the best possible integration with all the Hadoop features, the map and reduce functions should be written in Java, as that is also Hadoops programming language. There is also an alternative, which is called streaming. It means that the map and reduce functions interact with the framework with standard input and output streams. Streaming makes it possible to write the functions in any programming language.

Hadoop can read input from a local or network filesystem, but also its own distributed filesystem, HDFS. HDFS can store multiple copies of very big files on multiple nodes in a cluster. Compression can be used to save some space and time in transferring the files. Hadoop can automatically decompress the files when reading items for the map and reduce functions.

The Reduce step is heavy, as the framework first needs to group together all the values that have the same key into one node. Hadoop can optimize this step by using a special combine function, which is basically the same function as the reduce function, but it runs on each node with those items that are locally available. The combiner runs before the reduce function, acting as a kind of a pre-reduce and makes the reduce step lighter. The combiner, however, can not

be used in cases where the reduce function expects to have all the items at it's disposal at the same time.

One of Hadoops strengts is its strong focused ecosystem. There are a lot of additional supporting software and also commercial products that are based on Hadoop. Amazon Web Services (AWS), for example, have released an easy to use service called Elastic MapReduce. It allows to provision a cluster of the desired size and start a new MapReduce job on the cluster with a few clicks.

In the following example we used a cluster of 15 virtual computers from AWS to process 600GB of raw text data. The data was constituted of millions of books in english, organized in n-grams. N-grams means word sequences of N-words. In this case 3-grams were used. This data, provided by the Google n-gram project, was structured in lines as follows:

```
flowing from the 2008 46 45 44
```

Each line contains one n-gram from some year. The following numbers inform about the number of overall occurences, in how many pages it occurred and in how many books the ngram occurred. With this data, and a simple map function, we could find out what humankind considered fun in each decade since 1870. The results are given in the following listing:

- in 1870's anything was fun (7)
- in 1880's life was fun (4)
- in 1890's life was fun (6)
- in 1900's everything was fun (15)
- in 1910's life was fun (10)
- in 1920's everything was fun (9)
- in 1930's life was fun (18)
- in 1940's work was fun (45)
- in 1950's reading was fun (110)
- in 1960's reading was fun (144)
- in 1970's sex was fun (228)
- in 1980's sex was fun (189)
- in 1990's learning was fun (513)
- in 2000's learning was fun (597)

The number in paranthesis shows the number of books where the phrase was mentioned. The results can be debated, but they surely are interesting. Most interesting is perhaps the simplicity of the processing algorithm, of which the essential part is presented in the following listing:

```

public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
               Reporter reporter) throws IOException {
    String line = value.toString();
    String[] tokens = line.toLowerCase().split("\\s");
    if(tokens[1].equals("is") && tokens[2].equals("fun")){
        word.set(tokens[0] + "." + tokens[1] + "." + tokens[2] + "." + tokens[3]);
        IntWritable count = new IntWritable( Integer.parseInt(tokens[6]) );
        output.collect(word, count);
    }

    public void reduce(Text key, Iterator<IntWritable> values,
                     OutputCollector<Text, IntWritable> output,
                     Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

```

The map function did most of the work by scanning all of the 600GB data and emitting only a handful of lines forward for the reduction step. The reduce function counted together identical phrases. After the map and reduce functions were run, the remaining dataset was only a few megabytes of size, and the final processing could be done in a few seconds using a simple python script on a local computer. The map function took about 2.5 hours to process the 600GB dataset a cluster of 15 virtual computers.

4 Viewing Documents with CouchDB

CouchDB is a document database designed for the web. It builds on the RESTful[2] design principles and uses technologies such as JSON and JavaScript that are familiar to web developers. Documents are stored in JSON format and single documents can be requested as they are with an ID, but custom views are generated in the CouchDB server with map and reduce functions that are written in JavaScript.

CouchDB implements a highly optimized version of MapReduce. It allows to run a map function and then optionally a reduce function. The input data is split by documents, so each document gets through the map function separately. The idea with the map function is to index the documents by some key. The key may contain many values that then pass through the optional reduce function.

This process is optimized so that after the whole index is generated once, new documents do not require to map the old documents. Indexes are based on the keys, and once generated, the client can ask for the keys in certain order, or a certain range of the keys.

//Example of two users as JSON documents:

```
{
  "name": "joonas",
  "payments": {
    "2012-01": 10,
    "2012-02": 10,
    "2012-03": 10
  }
}
{
  "name": "poonas",
  "payments": {
    "2012-01": 9,
    "2012-02": 3,
    "2012-03": 5
  }
}
```

With this example document, we can think about a few useful map functions. The map may for example emit the document as it is, but with an added total payments field, which can easily be calculated with JavaScript. Another map could emit items where the key is payment month and the value is the amount of payment in that month. This would be done for all documents and then reduce function could count all the payments together, resulting in an index where the client can quickly look up the total sum of payments for each month.

```
// total payments for each user
function(doc) {
  total = 0;
  for( id in doc.payments ){
    total += parseInt( doc.payments[id] );
  }
  emit(doc.name, total);
}
```

Document databases are being developed enthusiastically, and one side-effect of that is fragmented ecosystems. There is one important fork of CouchDB, BigCouch, that scales CouchDB to multiple nodes. It provides the same API but the underlying software is a distributed system. This way the map and reduce functions can be run on parallel at different computers to optimize big views. There is a plan to merge BigCouch back to CouchDB repository, and hopefully more of that potential will be worked out.

5 Image Processing with MashReduce

MashReduce is a project to adapt MapReduce for building web mashups in short time. The rational is, that web data is distributed, and can thus be processed in parallel. If we want to scale up the mashups by using more content, or heavier algorithms, then finally the only alternative is to paralellize the computation.

In MashReduce, the map and reduce functions are usually used to process image data. In practice, the key-value items are such, that the value contains a jpeg-picture. Map functions can for example transform the image somehow or extract some information out of it. Reduce functions can for example create a collage of images in the same location. MashReduce also extends the MapReduce pattern by adding another type of function, called mashup. Whereas a map function is processed for each item and a reduce function is processed for each group of items, the mashup function is processed for all items together.

To run some task, the functions are first organized into a pipeline, which shows clearly all the steps of the task. The pipeline is then distributed to the available worker nodes and fed some input. The framework supports a few popular web content sources, such as Flickr and Picasa, where public albums can be automatically downloaded. The following picture illustrates the creation of a processing pipeline (right) from the available map, reduce and mashup agents (left).

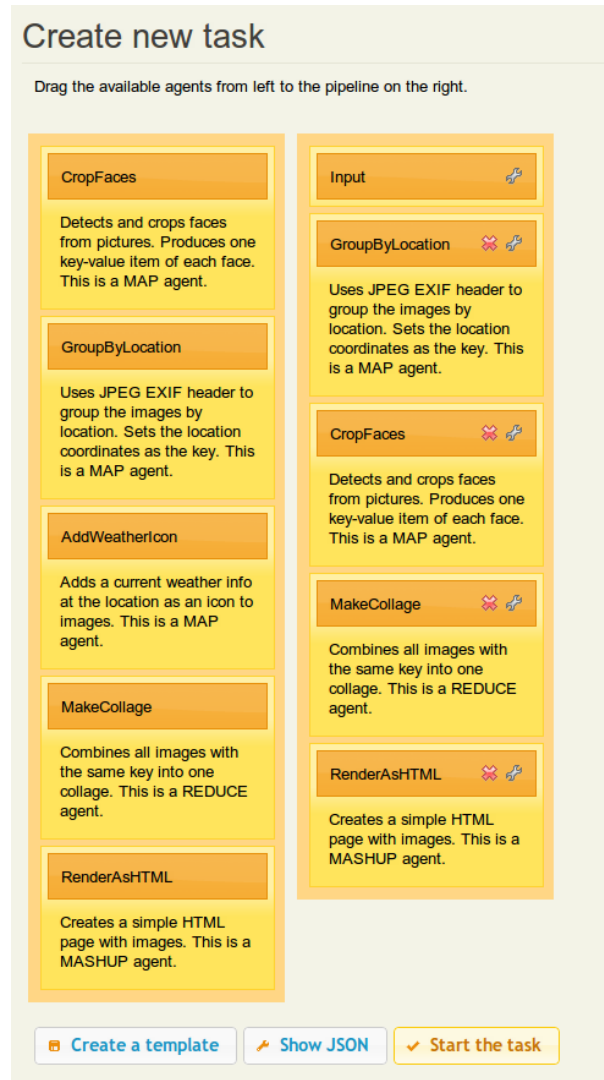


Fig. 2. Screenshot: Pipeline with map, reduce and mashup agents

This particular pipeline can download and process 22 pictures from three content sources using four worker nodes in less than 7 seconds. The result is a HTML-page that shows collages of faces taken in various locations.

One clear advantage of using the MapReduce pattern here is that each map, reduce or mashup agent exists as a self-contained module. They can be later combined in different ways for different application needs. The agents can take parameters, making them even more reusable. New applications can be created

faster, if it is possible to rely on already existing agents and combine them into new pipelines with new agents.

There is, however, some hidden information behind each agent, which limits the way they can be combined. The designer needs to get familiar with the existing agents to understand what kind of input they expect. For example, the GroupByLocation agent expects that the input images contain EXIF headers that tell the location where the picture has been taken.

Following is an example agent in python. This particular agent is used for extracting faces from images and emitting only each face forward to the next step. The agent relies on OpenCV library for the hard work using python bindings.

```
class Facedetection(MapBase):
    def init(self):
        self.hc = cv.Load("haarcascade_frontalface_alt.xml")

    def map(self, key, value):
        pil_img = Image.open(StringIO(value))
        cv_img = cv.CreateImageHeader(pil_img.size, cv.IPL_DEPTH_8U, 3)
        cv.SetData(cv_img, pil_img.tostring(), pil_img.size[0]*3)
        cv.CvtColor(cv_img, cv_img, cv.CV_RGB2BGR)

        if cv_img.width > 1280:
            width = 768
            height = int(float(width) / float(cv_img.width) * float(cv_img.height))
            small = cv.CreateImage((width, height), cv_img.depth, cv_img.nChannels)
            cv.Resize( cv_img, small )
            cv_img = small

        faces = cv.HaarDetectObjects(cv_img, self.hc, cv.CreateMemStorage(), 1.1)
        for (x,y,w,h),n in faces:
            cv.SetImageROI(cv_img, (x,y,w,h))
            cropped_face = cv.CreateImage((w,h), cv_img.depth, cv_img.nChannels)
            cv.Copy( cv_img, cropped_face )
            resized_face = cv.CreateImage((128,128), cv_img.depth, cv_img.nChannels)
            cv.Resize( cropped_face, resized_face )

            output = StringIO()
            cv.CvtColor(resized_face, resized_face, cv.CV_BGR2RGB)
            pil_img = Image.fromstring("RGB", cv.GetSize(resized_face),
                                      resized_face.tostring())
            pil_img.save(output, "JPEG", quality = 95 )
            self.emit( key, output.getvalue() )
            cv.ResetImageROI(cv_img)
```

6 Acknowledgement

I want to thank participants of VikingPlop 2012 for their input and especially Veli-Pekka Eloranta for shepherding this paper.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. OSDI p. 13 (2004)
2. Fielding, R.T.: REST: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine (2000), <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

Towards a Pattern Language for Teaching in a Foreign Language

Christian Köppe and Mariëlle Nijsten

Hogeschool Utrecht, Institute for Information & Communication Technology,
Postbus 182, 3500 AD Utrecht, Netherlands
{christian.koppe,mariëlle.nijsten}@hu.nl
<http://www.hu.nl>

Abstract. Teaching a technical subject in a foreign language involves more than just using a different language; there are specific problems related to the integration of content and learning. This paper begins with the mining of patterns which address these problems and intends to offer practical help to teachers by working towards a pattern language for technical instructors who teach students in a foreign language, and who are not trained in language pedagogy. We describe the background of teaching classes in a foreign language and propose the first two patterns which can be used when in this situation: LANGUAGE STATUS QUO and CONTENT-OBLIGATORY LANGUAGE.

1 Introduction

Education in different languages can serve different educational and social goals [7]. In today's competence-based educational systems, the focus is not on language learning as a separate subject, but on integrated learning with professional activities at its core, a so-called learning by doing approach. In such a context, teachers easily decide on the use of a foreign language as the medium of instruction, without giving the risks this may bring much further thought. Yet integrated learning can only be successful when sufficient attention is paid to both sides of the coin: language and core competences. So raising awareness among teachers to pay attention to both aspects is a first step and offering them a practical helping hand a second step. There are practices which can be used to implement these principles, but language pedagogy methodologies do not include the descriptions of these best practices detailed enough for subject-specific instructors to apply them without having had a language pedagogy training. Such teachers need a practical and easily applicable manual for a thorough integration of a foreign language in their courses.

This paper is part of a larger project for developing a pattern language for teaching in a foreign language. Along with this paper, which gives an introduction to the topic and describes the target group and the main focus of the patterns, two additional papers are available: one describing integrated language learning (helping technical instructors develop courses in a foreign language, using integrated language learning) and 4 patterns [5], and a second paper focusing on enhancing foreign language proficiency through integrated learning (how to improve advanced students' language proficiency while teaching in a foreign language) and three more patterns [6].

2 Target group

The proposed patterns are aimed at subject-specific teachers without any formal language pedagogy training who, as non-native speakers, occasionally teach a subject-specific class in a foreign

language to non-native speakers of this foreign language. Without knowing the specifics of language teaching, content teachers are often not aware of any language barriers when switching to a foreign language as a teaching medium. As the foreign language to be used usually will not be his/her mother tongue, the teacher may have some difficulty in expressing him/herself properly as well. For this specific target group, most best practices advised in language pedagogy handbooks lack solid underpinning arguments, the proper context of use, or an exact description of the problems they address. Without specific modern language pedagogy training, they need more clear rules of thumb to properly apply the right strategies.

3 Main pattern focus

The main focus of this work is to help instructors teach their course in a foreign language without the risk of students falling behind due to low language proficiency. A crucial question for instructors when the course is finished, is: when students fail their tests, is it because of lack of understanding of the technical content or lack of understanding of the second language used? It is obvious that students already have to know the foreign language at a certain level, otherwise the teaching would have to focus too much on language and not enough on the content. In higher education, these knowledge levels can be quite diverging due to various backgrounds. This diversity is the reason why there still are (or can be) language barriers. These barriers can lead to problems with comprehension of the technical content due to the language problems. To prevent failure due to language barriers, teachers can take precautionary measures. So the patterns aim to create awareness and help teachers design the right course for each audience on two levels: content as well as language.

This work therefore aims at reaching the following goals:

- Find and describe the common principles of existing methodologies and approaches in order to build a common vocabulary — a pattern language — which can be applied to generate the desired results.
- Provide teachers who are going to teach specific subjects in a foreign language with practical strategies to take language-related aspects into account without having to fully master a high-level language pedagogy methodology, such as Content Based Instruction (CBI) and Content Language Integrated Learning (CLIL).

4 Pattern language overview

The following list gives an overview of the patterns for teaching content in a foreign language¹. It can be also used as starting point for working with this language.

Patterns aimed at basic understanding of course content:

- Be aware that you are the TEACHER MODEL to students.
- Assess students' language proficiency in class room English — or another foreign language — as well as content specific jargon, this is the LANGUAGE STATUS QUO.
- Ensure that the material used for the course is at an appropriate language level by means of careful INPUT SELECTION.
- Define the CONTENT-OBLIGATORY LANGUAGE.

¹ As this pattern language is a work in progress, this list is not complete.

Layer	Layer Description	Content-Focused	Language-Focused
D	A part or unit of the curriculum as e.g. a course lasting one semester, time frame of all hours for that module (25 to 900 hours)	LANGUAGE STATUS QUO, INPUT SELECTION, CONTENT-OBLIGATORY LANGUAGE, LANGUAGE MONITOR, TEACHER MODEL	CONTENT-COMPATIBLE LANGUAGE
C	A thematic unit, time frame of one to many hours		LUCKY LANGUAGE CLOVER
B	A pedagogical unit, or a learning/teaching situation, time frame of several minutes to few hours		METATALK
A	A direct (pedagogical) interaction, time frame of several seconds to minutes		COMMENTED ACTION

Table 1. The layers of educational actions as defined by Baumgartner [2] and the corresponding patterns.

- Establish whether or not students indeed comprehend the course content and whether or not their command the foreign language improves by installing a LANGUAGE MONITOR.

Teachers who wish to enhance advanced students’ proficiency in a foreign language are referred to the following patterns:

- Include speaking and listening as well as reading and writing during classes and in assignments, these are the four leaves of the LUCKY LANGUAGE CLOVER.
- Define the CONTENT-COMPATIBLE LANGUAGE (in addition to the CONTENT-OBLIGATORY LANGUAGE).
- Urge students to reflect on the way they talk and write by stimulating METATALK.
- Use enhanced language and synonyms when giving explanations using COMMENTED ACTION and the earlier defined CONTENT-COMPATIBLE LANGUAGE.

Table 1 provides an overview of the patterns and the impact their application has on a course. The patterns in layer A can easily be applied in a single lesson, whereas patterns in layer D need more consideration as they generally comprise a full course. Content focused patterns should be applied before attempting to use any language focused patterns.

5 The Patterns

These patterns were mainly mined in existing literature and experience reports. Therefore they often miss sufficient known uses and fail at this moment to follow the *rule of three*. However, we intend to include more known uses after these patterns have been applied and adjusted more broadly and make them available as a whole pattern language.

The patterns use a version of the Alexandrian pattern format, as described in [1]. The first part of each pattern is a short description of the context, followed by three diamonds. In the second part, the problem (in bold) and the forces are described, followed by another three diamonds. The third part offers the solution (again in bold), the (empirical) background, consequences of the pattern application — which are part of the resulting context — and a discussion of possible implementations. In the final part of each pattern, shown in *italics*, we present some known applications.

In the following sections we present the following patterns: LANGUAGE STATUS QUO and CONTENT-OBLIGATORY LANGUAGE. After that the short versions of the other patterns — the patlets — are presented, the full patterns are published in [5, 6].

LANGUAGE STATUS QUO

You assume that students are at a sufficient level of general foreign language competences, i.e. they have knowledge of basic common vocabulary and grammar and can use the language. You now want to start teaching a course in this foreign language, with a foreign language as a medium of instruction so as to improve the use of this language in a professional setting.



Without knowing the actual level of foreign language competences of the students it is likely that the language parts of the course design are either too difficult for the students which hinders them in grasping the content or are too simple for them which means that their language understanding probably does not improve.

In undergraduate or graduate programs, students will often have different levels of language proficiency due to their different backgrounds.

Cultural Background. Some courses are taught in a foreign language — often English — to attract students from abroad and enable them to take part. The level of language proficiency and the way these students have learned the language in their original countries affects the way the foreign language can be used in such a course. A second background issue may be the students' level of academic English which can vary greatly, depending on the type of education students have had before entering the course and their knowledge of academic language in their first language [3, 10].

Available Material. In many domains, like e.g. ICT related subjects, the written classroom materials used are often available in English only, making it harder for instructors to teach these subjects in their mother tongue, as it forces teachers and students to translate parts of the texts used into the mother tongue, e.g. when giving explanations or answering questions. This results in poor quality translations and negatively affects both the teaching and the learning process.

Educational Career. Many courses (or studies) define minimal language requirements, like language courses which have to be attended and finished. This just gives an indication of the minimum level a teacher can expect from the students and still does not say anything about the variety of language levels present among students.

Standard Language. Even though it shows that the understanding and general knowledge of the foreign language is at a sufficient level, it still can lead to problems. In technology courses, the content contains a lot of specific terms — jargon is used as well as language-structures with content-specific semantics. Knowledge of this vocabulary is not reflected by standard language certificates.

Context-specific pretests. Pretesting is often aimed at a narrow range of aspects: what do students know on the subject? What is their general language proficiency? What is needed for competence based learning or integrated learning, is a specific pretest on communication competences used while performing selected professional and educational tasks in a specific branch or sector you train students for. Context-specific pretests are often tailor-made, though their components may be selected from existing proficiency tests.



Therefore: Get to know the language level of all students at the start of a course to obtain a realistic overview for your specific professional and educational goals. Use appropriate tests that include both general language competences and context specific linguistic competences, such as class room language, formal academic language, and core professional activities in your field. This is the basis for an adequate language integration in the course design.

The LANGUAGE STATUS QUO is usually gathered by one or more tests and should cover the aspects relevant for the course at hand. These aspects can include:

1. *general language competences* — Grammar and general vocabulary, but also reading, listening, writing, and speaking.
2. *content-specific language competences* — Knowledge of the course domain language, like jargon or often used language constructs etc. The CONTENT-OBLIGATORY LANGUAGE and the CONTENT-COMPATIBLE LANGUAGE can be used for testing these aspects.
3. *language-related competences* — Like giving (or *daring* to give) presentations in the foreign language, discussing problems in the language, speaking the language in front of a group, or creating formal writings in the foreign language.

The first aspect can be covered by looking at which courses in the foreign language the students already followed or the language certificates the students own. It is helpful to use proficiency tests based on international standard frameworks for language examination, such as the International English Language Testing System (IELTS), Common European Framework of Reference for Languages (CEFR) or Association of Language Testers of Europe (ALTE). But, as described earlier, there are more aspects which are (usually) not covered by such tests.

In some cases it can be useful to determine the content-specific language competences of the students, if e.g. the students in a course have different educational backgrounds. If the students follow a fixed study scheme and it can be assumed that they have a more or less equivalent level of language knowledge, this aspect can be omitted.

Commonly used proficiency or placement tests (often) do not include the competences to use the language in different educational and professional contexts. Missing these competences can lead to situations where students are not able to give a presentation because they have trouble speaking in front of groups in the foreign language (and not because they don't understand the content). These tests (often self reflective) give a clearer picture on how students apply the language.

Knowing the students' current levels per aspect forms the basis for an appropriate set-up of language elements in a course. Depending on the relevant aspects the following consequences can be identified:

- Difficulties in grammar and general language competences can be improved by promoting METATALK and including the missing parts in the CONTENT-COMPATIBLE LANGUAGE.
- Depending on the strengths and weaknesses of the students' competences to read, listen, write or speak the foreign language, an accordingly balanced mix of the four leaves of the LUCKY LANGUAGE CLOVER should be included in student activities.
- If content-specific language competences already are present, then the CONTENT-OBLIGATORY LANGUAGE and the CONTENT-COMPATIBLE LANGUAGE can be adjusted to cover a broader or deeper range of language aspects. Another consequence could be that less exercises need to put the focus in both content and language aspects.
- If content-specific language competences are not (sufficiently) present, then the missing parts of the CONTENT-OBLIGATORY LANGUAGE and the CONTENT-COMPATIBLE LANGUAGE should be taken into account during course design or course adjustment. Exercises

should expose the students repeatedly to these language aspects in different ways, e.g. by letting them research the meaning of different content-specific words, using COMMENTED ACTIONS during lectures and working groups, or let them give presentations which require the knowledge and usage of content language (and also makes use of the LUCKY LANGUAGE CLOVER).

- If the students have shortcomings in language-related competences then include exercises which let them develop and practice these competences.

De Graaf et al. suggest that students should be exposed to input at a (just) challenging level [4]. In order to determine this level, knowledge of the LANGUAGE STATUS QUO is required. This can also be used as the first check of the LANGUAGE MONITOR. The following checks can then be compared with the beginning situation.

One advantage of testing is that students will become more aware of their language proficiency and that they are able to determine themselves whether their language competences need further improvement. A disadvantage of applying this pattern is that it requires more work from the teacher and also extra time from the students.

The authors applied this pattern at the beginning of a course which was taught in English to students whose mother tongue was Dutch. They had to fill in a short survey stating their last followed courses in English and the grades received for those. The levels of these courses were known to the authors, making it easy to relate the grades for these courses to levels of international standard frameworks like the Common European Framework of Reference for Languages. Furthermore they were also asked to fill in a self-evaluation about their competences and ease of giving presentations, reading technical documentations, explaining technical problems, etc. The test showed the most students were afraid of giving presentations in English at the beginning of the course, so the amount of exercises and assignments which required student presentations was increased, starting with just giving small presentations about a small-scoped problem and ending with a presentation of their final project result.

CONTENT-OBLIGATORY LANGUAGE

The content of a course is mostly focused on one domain, which often has specific terminology used in this domain. If students have a low general language proficiency, the chances of them failing to understand the real meaning of this terminology increases [9].



Some lexical items and terminology of the foreign language are so closely related to the content of a course that mastering them is crucial to students in order to achieve the course objectives.

Some students may get the wrong understanding of the domain of the course contents. When explaining this terminology in the foreign language, they use phrases in the foreign language without grasping their meaning.

Definition Repetition. Students know that it is sometimes sufficient to memorize definitions without understanding, as tests are often asking for memorized knowledge only. Not all things have to be understood more deeply and in the broader context.

False Friends. When reading a text in a foreign language there are often words which are unknown or the meaning is only vaguely known. Usually the meaning becomes somewhat clearer in the broader context and through the position of the words, but these are just assumptions. Especially terminology in specific domains can give a different meaning to common known words, which can lead to “false friends”.



Therefore: Define the content-obligatory language before and during course design. Expose the students to this language continuously in different ways with an emphasis at the beginning of the course. Let this language repeatedly come back during the whole course to improve assimilation and understanding of this language.

The content-obligatory language can consist of different parts:

- vocabulary - the terms used in, and specific for, the domain covered in the course. Example for mathematics would be the terms: Subtraction, Addition, Division, and Multiplication.
- language constructs - domain-specific ways of using the language, which are specific for the domain. Examples are the mathematical constructs: “x is subtracted from y” or “factor out the greatest common factor”.

The language specific for a domain often includes visuals as well, but these are mostly independent of the language used and should therefore already be included in the content-related material. However, these are also very helpful in language acquisition [4], as they help in relating knowledge structures to associated language expressions [7]. In some cases it therefore can be considered to be helpful if these visuals are also explicitly added to the content-obligatory language. The known uses section gives an example of this.

The defined language is a reference for the course design — the used materials, presentations, etc. It therefore also forms the basis for INPUT SELECTION.

Use different communication ways — as defined by the LUCKY LANGUAGE CLOVER — for explaining the language and exposing students to the language and letting students grasp, practice and apply the language.

To check whether they really understood the domain concepts, ask students to explain them in their own words. This way you will find out whether they've simply learned phrases or really grasped a deeper understanding of the concepts and terminology. Especially in the beginning these explanations can also be in the students' mother tongue, which gives more insight into the deepness of understanding [8].

De Graaf et al. suggest the learners should be stimulated to request new vocabulary items [4]. However, in order to ensure that these items contribute to the content too, a list with the essential vocabulary items should be made in advance. This list could also contain the items of the CONTENT-COMPATIBLE LANGUAGE.

A serious risk when defining the CONTENT-OBLIGATORY LANGUAGE is that it costs the teacher a lot of preparation time, mainly for two reasons: (a) it is not always obvious which parts of the language are really essential for understanding the course content, so determining these parts requires extra time and (b) the amount of relevant terms and constructs can be overwhelming, leading to excessive lists which are not easy to create and handle. To avoid this, select input texts of the appropriate level, using the INPUT SELECTION pattern or use visuals to explain domain language.

Köppe and Weber defined in a course on (Design) Patterns & Frameworks a list with the essential terms (and important visuals like UML class- and sequence-diagrams), which included: class, association, generalization, specialization, inheritance, interface, Pattern, Framework, sections (of a pattern), context, forces, abstraction, coupling etc. Even though some of these terms can be assumed as known to the students because of earlier programming and UML courses, they are included again here as they are essential for understanding the patterns but also the pattern solutions and their impact on the overall design. In the first lecture the students had to give short presentations in English about general design principles and techniques including examples. The use of some of the terms was implicitly required for this exercise, and the teachers emphasized the importance of these terms by immediately asking questions like "Why have you chosen a generalization and not an association" which requires deeper understanding of the terms in order to be answered correctly. This exercise gave a good overview of which terms the students already knew at a sufficient level and which terms did need more explanation and repetitive attention during the following lectures.

Nijsten designed an energizer exercise that took around 10 minutes. She made flash cards with an English expression — chosen from the CONTENT-OBLIGATORY LANGUAGE — on one side and its definition in English at the back side. Students walked around in class. Whenever they ran into another student, they showed their flash card with the definition up front and asked for a definition in English (or the mother tongue). The fellow student then returned the question using his flash card. Afterwards, they exchanged cards and walked on to have such language exchanges with other class mates. This exercised lifted spirits and improved students' mastery of the main course vocabulary as well.

The patlets of the other patterns

INPUT SELECTION

Most courses make use of material — literature, websites, tutorials etc. — which covers the content of the course. You have identified both the CONTENT-OBLIGATORY LANGUAGE and most parts of the CONTENT-COMPATIBLE LANGUAGE and you know the LANGUAGE STATUS QUO of the students' language levels. You now want to start to look for the material.



Available material often differs in both language levels and comprehensibility, and can be too difficult or too easy for students. Both cases will lead to problems during the course.



Therefore: select comprehensible course input that explains the subject matter in a way that matches students' language levels and interests.

CONTENT-COMPATIBLE LANGUAGE

You identified the CONTENT-OBLIGATORY LANGUAGE and included in the course design opportunities so that the students can master it. However, most domains contain more elements of a specific language: synonyms, proverbs, expressions, phrases, metaphors, etc..



Only mastering the obligatory language of a course's content might be sufficient to fulfill the course's requirements, but it limits the students in their expressiveness and does not improve the overall quality of students' language skills.



Therefore: Identify the language constructs and expressions of the course domain which are additional to the obligatory elements. Include opportunities for learning these in your course design and course execution.

LUCKY LANGUAGE CLOVER

You are thinking about the tasks you want to include in the course design and want to ensure that they also cover the CONTENT-OBLIGATORY LANGUAGE and the CONTENT-COMPATIBLE LANGUAGE aspects appropriately.



Exposing the students to language comprehension only — reading and listening — is not sufficient for improving production levels of the foreign languages.



Therefore: Include all four types of linguistic competences in your course design. Promote reading and listening, but have students writing and speaking in the foreign language as well.

METATALK

Students understand the content and are using the foreign language, making use of all four leaves of the LUCKY LANGUAGE CLOVER, but the language competences of the students still vary.



Students are not aware of their foreign language shortcomings and keep using incorrect language constructs and terms.



Therefore: Stimulate foreign language learning by including exercises or other appropriate course parts which require a collaborative reflection on language usage.

LANGUAGE ROLE MODEL

You are asked to give a course in a foreign language.



Learning is also imitating, but imitating incorrect language usage of a teacher will affect the students' learning of the language negatively.



Therefore: make careful language preparations to ensure that you can instruct students using a foreign language in a correct way for all related language parts. Use the language during the course always as correct as possible.

COMMENTED ACTION

Also known as: Think Aloud Protocol, Show and Tell.

Often in courses or lectures you show or demonstrate some content-specific activities. You are aware that you are a LANGUAGE ROLE MODEL and have identified the CONTENT-OBLIGATORY LANGUAGE and the CONTENT-COMPATIBLE LANGUAGE of your course. Students do not yet know these language constructs or were just introduced to them.



If the students only see the activities done by the teacher, they might be able to execute them themselves, but will have difficulties describing in the foreign language what they are doing. Their vocabulary and expressiveness will not increase.



Therefore: Do not only show or demonstrate complex abilities but give a spoken description of the steps you are taking. Use the earlier identified language terms when you show their meaning.

LANGUAGE MONITOR

Also known as: Formative Assessment.

You have designed a course with a focus on both content and language, identified the CONTENT-OBLIGATORY LANGUAGE and the CONTENT-COMPATIBLE LANGUAGE and used this for INPUT SELECTION. You have chosen specific learning activities to stimulate METATALK, support language learning with COMMENTED ACTIONS and are aware that you are a LANGUAGE ROLE MODEL. You now want to assess whether your learning activities have had the expected result: an improvement of the students' foreign language skills.



Judging the progress students make with language acquisition is not possible during lecturing, as this already requires all effort of the teacher. But without judgement you don't know if the students make progress with language acquisition.



Therefore: Implement regular assessments on the language skills of the students to determine whether they grasped the content and whether their language skills have improved, and use these outcomes to intermittently adapt your course.

Acknowledgements

We want to thank our VikingPLoP 2012 shepherd Neil Harrison. He helped us with his deep insights in patterns, his questions and comments led us to improvements which we probably had not made without him. We also thank the members of the VikingPLoP 2012 writers' workshop who gave constructive feedback on this work.

References

1. Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, August 1977.
2. Peter Baumgartner. *Taxonomie von Unterrichtsmethoden: Ein Plädoyer für didaktische Vielfalt*. Waxmann Verlag, Münster, 2011.
3. Patricia L. Carrell and Joan C. Eisterhold. Schema Theory and ESL Reading Pedagogy. *Tesol Quarterly*, 17(4):553–574, October 1983.
4. Rick De Graaff, Gerrit Jan Koopman, and Gerard Westhoff. Identifying Effective L2 Pedagogy in Content and Language Integrated Learning. *Vienna English Working Papers*, 16(3):12–19, 2007.
5. Christian Köppe and Mariëlle Nijsten. A Pattern Language for Teaching in a Foreign Language - Part 1. In *Preprints of the 17th European Conference on Pattern Languages of Programs, EuroPLoP 2012*, Irsee, Germany, 2012.
6. Christian Köppe and Mariëlle Nijsten. A Pattern Language for Teaching in a Foreign Language - Part 2. In *Preprints of the 19th Pattern Languages of Programs conference, PLoP'12*, 2012.
7. Constant Leung. Language and content in bilingual education. *Linguistics and Education*, 16(2):238–252, 2005.
8. Peeter Mehisto, David Marsh, and Maria Jesus Frigols. *Uncovering CLIL: Content and Language Integrated Learning in Bilingual and Multilingual Education*. Macmillan Education, Oxford, 2008.

9. M Met. 7 Teaching content through a second language. *Educating second language children The whole child the whole curriculum the whole community*, 7(1994):159, 1994.
10. MJ Schleppegrell, Mariana Achugar, and Teresa Orteíza. The grammar of history: Enhancing content-based instruction through a functional focus on language. *Tesol Quarterly*, 38(1):67–93, 2004.

A Pattern Language for Dialogue Management

Dirk Schnelle-Walka, Stefan Radomski

Telecooperation Group

Darmstadt University of Technology

Hochschulstraße 10

D-64283 Darmstadt, Germany

`[dirk|radomski]@tk.informatik.tu-darmstadt.de`

phone: +49 (6151) 16-64231

March 2, 2012

Abstract

Modeling human computer interactions as dialog, while originating in voice user interfaces, is becoming increasingly important for multi-modal systems. Different approaches with regard to formalizing and managing dialogues exist with their specific strength and weaknesses. In this paper, we present existing dialogue management techniques as patterns to give a basis for decision support when developing interactive systems in different scenarios.

1 Introduction

Describing the user interface of graphical interactive systems today is predominantly achieved by applying the Model-View-Controller (MVC) pattern, or one of its variations. In this approach, the different screens of an application can be organized into graphical widgets as the view or presentation component. These widgets will trigger callbacks into a controller component for the various user interface events (e.g. `onClick` or `onMouseOver`). The controller, in turn, updates the underlying model of an application leading to changes in the view component again.

Generalizing this approach, one can conceive the view as a set of system supplied entities, enabling the user to generate certain interface events the system is prepared to handle [9]. In the context of graphical interfaces, this might be a box to enter some text or a list of items to scroll through; in the context of voice interfaces, this might be a set of utterances the system is prepared to recognize. The controller is associated with the view as it describes the systems reaction when one of the enabled user interface event is actually observed. The model is the formalization of all the applications

state required to generate the user interface and is modified in response to user interface events.

While the MVC pattern describes the system components and their responsibilities in performing a single iteration of a user interaction feedback loop (see fig. 1), dialogue managers are concerned with the overall organization of a dialogue as a sequence of coherent turns to achieve the users goal. As such, dialogue managers may employ the MVC pattern for a single turn, but their actual responsibility is to provide a coherent global structure of user interaction. The different approaches to arrive at such a coherent structure are the subject of the pattern language presented in this paper.

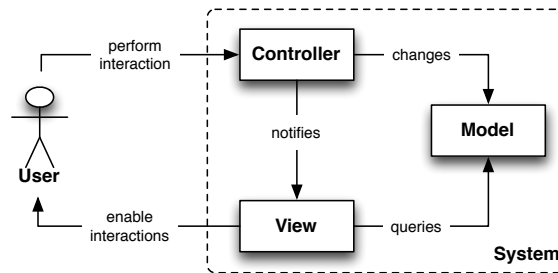


Figure 1: User interaction feedback loop in the MVC pattern.

The requirement for a component to ensure a coherent overall dialogue structure becomes obvious when we consider Voice User Interfaces (VUI) (see fig. 2). As the modality of speech is transient, there is no persistent view displayed to user and the system needs to maintain a discourse context as the set of shared beliefs and possibly intentions it identified during the course of interaction with a human user. Maintaining such a discourse context in the form of a dialogue manager can be beneficial not only for VUIs but for classical GUIs and especially multi-modal interactive applications as well.

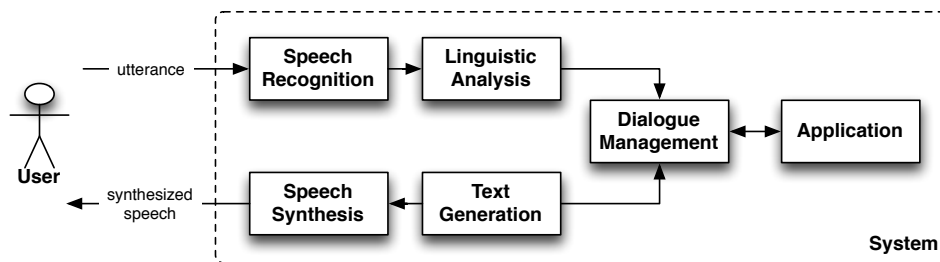


Figure 2: Architecture of an ASR system (from [6]).

1.1 Dialogue

There are different definitions of the term “dialogue”, some rather focussed on spoken dialogues, others with a broader focus on human computer interaction in general. The Merriam Webster dictionary¹ defines a dialogue as:

Dialogue: a conversation between one or more persons; also: a similar exchange between a person and something else (as a computer).

Conversation: an *oral* exchange of sentiments, observations, opinions, or ideas.

Another definition from an ITU-T Recommendation [5] defines dialogue as:

Dialogue: A conversation or an exchange of information. As an evaluation unit: One of several possible paths through the dialogue structure.

wherein conversation remains undefined. The definition possibly most in line with multi-modal interaction might be the one from Nielson [7]:

Dialogue: a recursive sequence of inputs and outputs necessary to achieve a goal.

Nielson himself remarks some problems with the definition e.g. that user input can not always be “chopped up into sets of discrete interactions”, a notion that still permeates all dialogue management techniques and becomes obvious e.g. when dragging an object.

1.2 Dialogue Management

There are again different definitions for *dialogue management* or a *dialogue manager* who is responsible to handle dialogue management, but they all are more or less in line with notion of executing dialogue descriptions to provide the user interface. Traum [18] defines a dialogue manager as follows:

A **dialogue manager** is that part of a system that connects the I/O devices [...] to the parts that do the domain task reasoning and performance.

Another definition from Rudnicky [11] defines dialogue management:

[**dialogue management**] provides a coherent overall structure to interaction that extends beyond a single turn[...]

The key notion here is the identification of the dialogue manager as a discrete subsystem of an application to handle the global user interaction. For every approach to dialogue management there is a corresponding formalization of dialogues as we will see in the patterns below.

¹<http://www.merriam-webster.com>

1.3 Dialogue Acts

There is something to be said about the granularity or level of abstraction of user interface events. We will need this abstraction in the patterns described later on. With classical GUIs, user interface events are usually classified by actions on widgets. For example, we have a class of user interface events for all clicks on a button and can get instance data by inspecting the representation of the event (e.g. the spatial coordinates or the index of a physical button on a pointing device). While this approach is also suitable for dialogue managers, we might also choose to abstract user interface events even further.

The most abstract representation that is still useful in operationalizing dialogue management is that of “dialog acts”. The concept originated as “speech acts”, introduced in the book “How to do things with words” from John Austin in 1962 [1]. Herein Austin identified several functions of utterances, such as “assertions” or “directives” to classify utterances with regard to their function in a dialogue. Applying the concept to other modalities, these acts can form the basic tokens for dialogue management, that is, a dialogue manager would only operate on such acts and components between the systems input devices and the dialogue manager would refine a set of user interface events into a dialogue act.

In that sense, dialogue acts are special speech acts. For instance *question* is a speech act, but *question-on-hotel* is a dialogue act. Consequently, speech acts are stable while dialogue acts may depend on the system.

2 Patterns

Patterns are an established way of conveying design knowledge for the design of user interfaces [2], guiding developers in the design of applications, e.g. for mobile devices [8] or multi-medial settings [10]. In the domain of voice user interfaces we build upon existing work from [14, 13, 12, 15, 16]. However, there is no such work we are aware of, for dialogue management. Existing overviews about the state of the art of dialogue management like [3] already gives a basic overview about this domain but does not use the pattern format that allows for an easier access to the information given. Moreover, he does not provide information of the applicability of the presented dialogue managers.

In this section, we address this shortage and describe a first set of dialog management patterns, helping developers to select an appropriate dialogue management strategy that fits their current design problem. An overview of the language with its relations is shown in figure 2. We consider the PROGRAMMATIC DIALOGUE MANAGEMENT as an anti-pattern with regard to dialogue management, as it does not support any dedicated feature for coherent dialogue behavior spanning more than one turn. Nevertheless, PRO-

GRAMMATIC DIALOGUE MANAGEMENT is the approach most people start with when developing interactive applications. The patterns are furthermore grouped with regard to who is experiencing the problem the pattern solves, that is the application developer or the end-user.

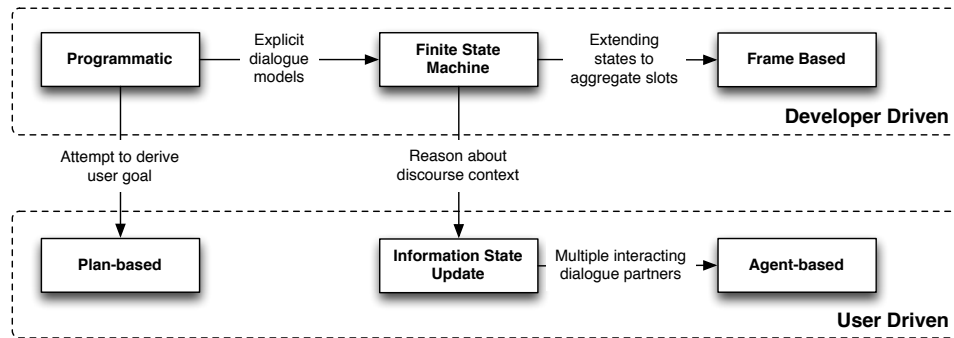


Figure 3: Overview of the pattern language

We basically stick to the format that we started in [14] with few adaptations. The format is based on the Coplien format [4] and also follows the suggested format of Tešanović [17] which we find to be useful to talk about design issues in human computer interaction.

PROGRAMMATIC DIALOG MANAGEMENT

Intent

Implement an interactive application with unimodal interaction and no need for explicit dialogue management.

Context

There is little or no need of decoupling dialogue structure and application logic. Suitable for straightforward, unimodal applications with only occasional changes to dialogue structure.

Problem

How to incorporate simple, unimodal user interaction into an application while still decoupling the user interface from the application logic?

Forces

- You want to separate application logic from the user interface presentation.
- You do not need to separate application logic from the dialogue structure.
- You do not need a coherent dialog structure beyond one turn.
- You do not care much about modifying the dialog structure later on.
- You expect your user to interact within a single modality.

Solution

The established solution to incorporate user input into an application is its separation into subsystems per MODEL-VIEW-CONTROLLER pattern. To implement this strategy, consider the following:

1. Identify classes of user input events (e.g. all clicks on a given button).
2. Provide entities to enable the user to generate input events suitable and meaningful to change the current application state.
3. Dispatch over the user input event's class and perform the associated operation on the application's state.
4. Unless the user generates an input event signaling her intention to quit interaction, goto 2.
5. The entities, enabling the user to generate input events can be collected into a reusable library.

Consequences

- ☺ You get first results and/or mockups rather fast.
- ☺ Application logic and the user interface are decoupled.
- ☺ The dialogue structure and its implementation are tightly coupled.
- ☺ There is no entity to guarantee a coherent global dialogue structure.

Known Uses

Every application without an explicit dialog model uses this pattern or a variation thereof.

Also Known As

This pattern is the MODEL-VIEW-CONTROLLER [1] pattern as it describes the components and their responsibilities in performing a single turn.

References

- [1] T. Reenskaug. Models - views - controllers. Technical report, Xerox Parc, 1979.

FINITE STATE DIALOGUE MANAGEMENT

Intent

Provide an interactive application with an easy way to adapt the dialog structure later on.

Context

The final dialog structure has not yet been identified or is likely to change fairly often.

Problem

With the dialogue structure implicit in the control flow, adaptations may have undesired consequences for the application logic.

Forces

- Changing dialogue structure should not affect application logic.
- Dialogue structure ought to be specified by user interface experts.
- Small variations of the dialogue structure may exist within different scenarios.

Solution

Decouple dialogue structure from application logic by expressing all variations of the dialogue as a finite state machine and implement the user interaction separately per state.

1. Model the dialog structure as a directed transition graph.
2. Provide a system output when a state is entered.
3. Expect user input.
4. Dispatch outgoing transition depending on user input.

By implementing the dialog as a transition over discrete interaction states, user interface experts can organize the overall dialogue structure, while the application programmers can provide simple per state interaction.

Consequences

- ☺ Decouples dialogue structure from control flow.
- ☺ Enables division of labour between programmers and user interface designers.
- ☺ Good general tool support.
- ☺ Local dialogue structure is obvious.
- ☺ Verbose descriptions lead to poor understanding of global dialogue structure.

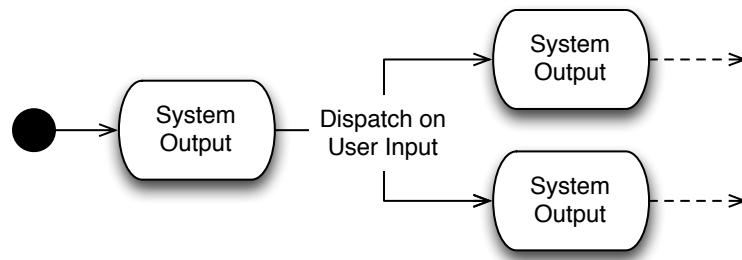


Figure 4: Finite state dialogue management.

- ☹ Tends to result in rigid dialogues.
- ☹ Requires dedicated runtime support to interpret dialog models.

Known Uses

- Directly supported by VoiceXML [3]
- Java Server Faces [2]
- Xcode Storyboards [1]

Related Patterns

STATE-DRIVEN TRANSITION FSM describes the basic mechanism of the underlying state machine[4]. FRAME BASED DIALOGUE MANAGEMENT allows for more flexibility in the user input.

Also Known As

FSM

References

- [1] Apple Inc. Cocoa Application Competencies for iOS: Storyboard. <https://developer.apple.com/library/IOs/#documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html>, 2010. Last accessed: February 2012.
- [2] Eric Jendrock, Ian Evans, Devika Gollapudi, Kim Haase, and Chinmayee Srivathsa. *The Java EE 6 Tutorial: Basic Concepts*, chapter JavaServer Faces Technology. Pearson Education, 2011.
- [3] Matt Oshry, R.J. Auburn, Paolo Baggia, Michael Bodell, David Burke, Daniel C. Burnett, Emily Candell, Jerry Carter, Scott McGlashan, Alex Lee, Brad Porter, and Ken Rehor. Voice Extensible Markup Language (VoiceXML) Version 2.1, W3C Recommendation. <http://www.w3.org/TR/voicexml21/>, June 2007.

- [4] Sherif M. Yacoub and Hany H. Ammar. Finite state machine patterns. In Jens Coldewey and Paul Dyson, editors, *Proceedings of the 3rd European Conference on Pattern Languages of Programms (EuroPLOP 1998)*, Irsee, Germany, July 8-12, 1998, pages 401–428. UVK - Universitaetsverlag Konstanz, 1998.

FRAME BASED DIALOGUE MANAGEMENT

Intent

Allow for adaptations of dialogue structure without altering application logic, but try to ease the verbosity of finite state dialogue models.

Context

Several pieces of information are related and form a frame that always has to be provided as a unit during interaction.

Problem

Modeling dialogues as simple transition graphs leads to very verbose and large dialogue descriptions that impede an understanding of the global dialogue structure.

Forces

- The global dialogue structure should remain comprehensible.
- Some information units naturally form compound information.

Solution

The solution extends the `FINITE STATE DIALOGUE MANAGEMENT` by abstracting the collection of frames into a new compound state. To implement this strategy, consider the following:

1. Identify related information as information slots in a frame that always occur in unison.
2. Abstract the collection of these slots into a new compound state.
3. Allow for arbitrary order of filling the slots by iterating within the compound state until all slots are filled. (Depending on the actual user interface, it might be possible to fill multiple slots in a single turn.)
4. Only use the compound state in the actual state transition graph.

Consequences

- ☺ Less verbose than FSM.
- ☺ More flexible within one frame.
- ☺ Good tool support with speech due to VoiceXML.
- ☹☹ Information slots within a frame always have to occur in unison.
- ☹ Inherits most drawbacks from FSM.
- ☹ Still rather verbose.

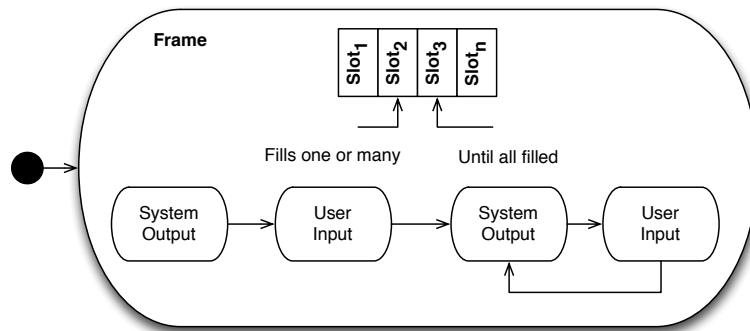


Figure 5: Frame based dialogue management.

Known Uses

- Directly supported as the form interpretation algorithm in VoiceXML [2]
- HTML forms [3]

Related Patterns

FRAME BASED DIALOGUE MANAGEMENT is an extension to FINITE STATE DIALOGUE MANAGEMENT where multiple information slots can be filled at once. This is related to the more compact representations of FSMs like Harel statecharts [1] as it is in essence a hierarchical view of FSMs with substates.

References

- [1] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987.
- [2] Matt Oshry, R.J. Auburn, Paolo Baggia, Michael Bodell, David Burke, Daniel C. Burnett, Emily Candell, Jerry Carter, Scott McGlashan, Alex Lee, Brad Porter, and Ken Rehor. Voice Extensible Markup Language (VoiceXML) Version 2.1, W3C Recommendation. <http://www.w3.org/TR/voicexml21/>, June 2007.
- [3] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification. <http://www.w3.org/TR/html4/>, 1999. Last accessed: February 2012.

INFORMATION STATE UPDATE

Intent

Allow for more flexible dialogues with a certain amount of *intelligence* in the dialogue structure.

Context

There is a formal knowledge-base pertaining to the domain of the application that enables reasoning and logical inference. As such, a dialogue can be conceived as shared reasoning between the user and the system.

Problem

When there is an established set of formal facts for a problem domain, human users expect a dialog system to utilize and reason about this information when performing a dialogue (e.g. not to establish each and every fact anew).

Forces

- An explicit representation of the dialogue structure as per FSM or FRAME-BASED is unsuitable due to the sheer amount of variations that would have to be considered.

Solution

Model a dialog as a set of rules with prerequisite and effect on a discourse representation structures. Within the prerequisites, logical reasoning is enabled as to conclude certain facts from the current discourse representation structures and the knowledge base.

The solution adopts the information state theory of Traum and Larson [2], wherein a dialogue is conceived as a set of transformation-rules of an information state due to observed dialog moves. Noteworthy is, that the system itself can react to its own information state updates and that system output is just a side-effect of the application of a given transformation-rule.

To implement this strategy, consider the following:

1. Formalize your information state as a set of entities and discourse representation structures.
2. Model your dialog as a set of transformation-rules for this information state. (Simple information slot filling with reasoning is already provided by Traum et al.²)

Consequences

- ☺ Grounding in dialog theories
- ☺ Way more expressive than FSM

²<http://www.ling.gu.se/projekt/trindi/trindikit/>

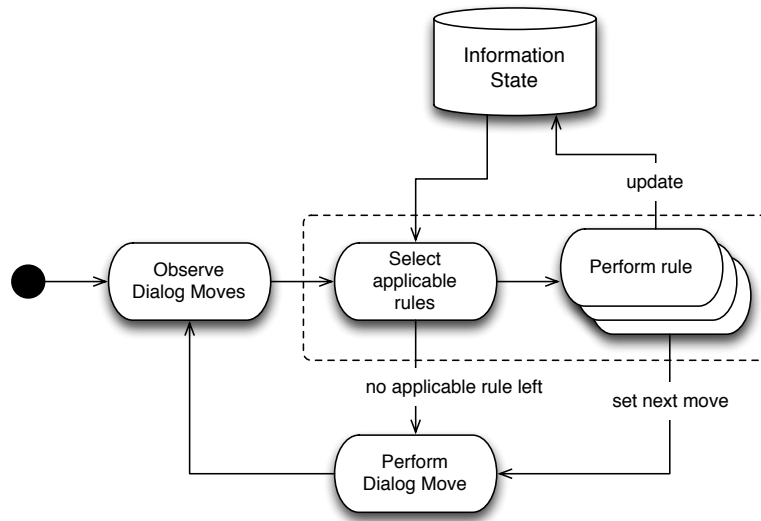


Figure 6: Information State Update dialogue management.

- ☺ Rules can be reused within a theory
- ☹ Opaque dialog behavior
- ☹ Runtime semantics a bit fuzzy (what happens if two rules specify moves)

Known Uses

- Traum et al. implemented their theory in the TrindiKit framework [2].
- Kronlid et al. implemented this approach with SCXML [1].

Related Patterns

The approach can be conceived as a huge FSM dialogue, wherein transitions are triggered not only by user input but also system events and logical reasoning is available.

References

- [1] F Kronlid. Implementing the information-state update approach to dialogue management in a slightly extended scxml. *Proceedings of the SEMDIAL*, 2007.
- [2] S. Larsson and D.R. Traum. Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural language engineering*, 6(3&4):323–340, 2000.

PLAN BASED DIALOGUE MANAGEMENT

Intent

Uncover the user's underlying goal to guide the actual dialogue management.

Context

The user has one of several clearly defined goals when approaching the system and the dialogue can be conceived as a sequence of dialogue acts [6] to achieve this goal.

Problem

The user is only interested in performing one dialog act from a subset of reasonable alternatives at a time to achieve his goal.

Forces

- Offering a wide range of options within a user-interface, decreases dialogue efficiency.
- Ambiguities in observed sequences of dialog acts can be interpreted as different goals.

Solution

Identify the actual user's goal early on, and decrease the number of available options accordingly. The plan based theories of communicative action and dialogue provide the basis for the solution [1, 2, 3] where it is the listener's task to detect the speaker's dialogue act and respond accordingly. To implement this strategy, consider the following:

1. User goals are ultimately sequences of dialog acts.
2. Observing sub-sequences of dialog acts can already help to predict the overall goal.
3. Use these predictions as an additional knowledge source for one of the other approaches.

Example

A typical example dialogue using this pattern could be

User: Where are the steaks you advertised?

Computer: How many do you want?

...

Consequences

- ☺ Can be very natural

- ⊖ Hardly operational
- ⊖ Very domain specific
- ⊖ Non-obvious application
- ⊖ Not an actual dialogue manager, more of a supporting approach.

Known Uses

- Within Verbmobil [4], the problem of translating meeting calls was considered and plan-based dialogue techniques were used to support the translation unit with regard to the current plan of the participants to resolve semantic ambiguities.
- Collagen [5] tries to find the closest predefined, hierarchical plan that matches the observed user actions. By changing the current plan via minimal extensions with regard to observed user actions, the system tries to conclude one of the predefined plans or eventually asks the user for clarification.

Related Patterns

Relies on STATE-DRIVEN TRANSITION FSM at the turn level [7].

References

- [1] J.F. Allen and C.R. Perault. Analyzing Intentions in Dialogues. *Artificial Intelligence*, 15(3):143–178, 1980.
- [2] D.E. Appelt. *Planning English Sentences*. University Press, Cambridge, 1985.
- [3] P.R. Cohen and H.J. Levesque. Rational Interaction as the Basis for Communication. In *Intentions in Communication*, Cambridge, Massachusetts, 1990. M.I.T. Press.
- [4] Susanne Jekat, Alexandra Klein, Elisabeth Maier, Ilona Maleck, Marion Mast, and J. Joachim Quantz. Dialogue acts in verbmobil. Technical report, 1995.
- [5] Charles Rich, Candace L. Sidner, and Neal Lesh. COLLAGEN: Applying Collaborative Discourse Theory to Human-Computer Interaction. Technical Report TR2000-38, Mitsubishi Electric Research Laboratories, Cambridge, Massachusetts, November 2000.
- [6] J.R. Searle. *Speech acts: An essay in the philosophy of language*. University Press, 1969.

- [7] Sherif M. Yacoub and Hany H. Ammar. Finite state machine patterns. In Jens Coldewey and Paul Dyson, editors, *Proceedings of the 3rd European Conference on Pattern Languages of Programms (EuroPLoP 1998)*, Irsee, Germany, July 8-12, 1998, pages 401–428. UVK - Universitaetsverlag Konstanz, 1998.

AGENT BASED DIALOGUE MANAGEMENT

Intent

Model interaction with distinct subsystems as agents with their own beliefs, desires, intentions (and obligations).

Context

There are several distinct subsystems each contributing to the overall dialogue behaviour and a facilitator to merge their intentions into a coherent system output.

Problem

Interacting with complex distinct subsystems often involves negotiating dialogue behaviour among several specialized system components. These components have to be organized in order to agree upon an observable dialogue behaviour.

Forces

- Loosely coupled subsystems contribute to the overall dialogue structure.
- Potentially conflicting intentions of subsystems need to be merged into a common system output.
- Interpretation of information state differs per subsystem.

Solution

Potentially conflicting goals of subsystems need to be negotiated to present a coherent dialog structure. This can be realized by modeling each subsystem as an agent with a formalized notion of (i) beliefs as the set of facts it holds true, (ii) desires as the goals it needs to achieve in order to perform, (iii) intentions as the actual system output it would like to see realized to further its own goals. In an extension of Traum et al. [1] it was proposed to explicitly model obligations as the things other agents expect from a given agent when their intentions are realized. To implement this strategy, consider the following:

1. Define each agent's initial desires (e.g. to collect some pieces of information).
2. Use the information-state approach within an agent to derive a set of intentions per observed information-state.
3. Use a meta-agent as a facilitator to realize turn-taking and agent selection.
4. Other agents update their beliefs and consequently their intentions by observing the selected agent perform.

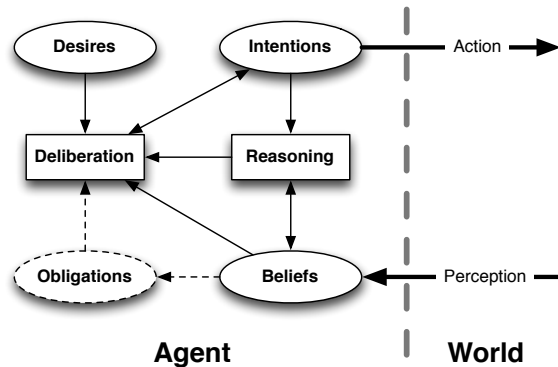


Figure 7: BDI model with obligations (adapted from [1]).

Consequences

- ☺ Adaptive dialogs
- ☺ Modeling of dialog conventions
- ☺ Collaboration between agents is expressible
- ☺ Opaque dialog behavior
- ☺ Depending on approach used in agent, inherits e.g. drawbacks of information-state update approach.

Known Uses

1. Trains-93 [1]

Related Patterns

INFORMATION-STATE UPDATE can be used within an agent.

References

- [1] Daniel Traum. Conversational Agency: The Trains-93 Dialogue Manager. *Proceedings of the Twente Workshop on Language Technology 11: Dialogue Management in Natural Language Systems*, pages 1–11, July 1996.

3 Conclusion

In this paper we introduced a first set of patterns for dialogue management. They integrate into the pattern language that we introduced in [14] and continued in [13, 12, 15, 16] by providing developers with a means to select the appropriate approach for multi-modal dialogue management. The patterns

we described are based on descriptions of dialog management in the past 30 years. We grouped the patterns with regard to “who is experiencing the problem the pattern solves”.

Patterns that are driven by the developer are: PROGRAMMATIC DIALOG MANAGEMENT can be used to implement an interactive application with unimodal interaction and no need for explicit dialogue management. This is de facto an anti-pattern with regard to dialogue-management.

FINITE STATE DIALOGUE MANAGEMENT provides an interactive application with an easy way to adapt the dialog structure later on.

FRAME BASED DIALOGUE MANAGEMENT as a variation of FINITE STATE DIALOGUE MANAGEMENT allows for easy adaptations of dialogue structure and tries to ease the verbosity of finite state dialogue models.

Patterns that are driven by the end-user are INFORMATION STATE UPDATE allows for more flexible dialogues with a certain amount of *intelligence* in the dialogue structure.

PLAN BASED DIALOGUE MANAGEMENT aim for uncovering the user’s underlying goal to guide the actual dialogue management.

AGENT BASED DIALOGUE MANAGEMENT express interaction with distinct subsystems as agents with their own beliefs, desires, intentions and obligations.

In the future we will extend our language by describing more recent variances of the described prototypes.

References

- [1] J. L. Austin. *How to do Things with Words*. Oxford University Press, New York, 1962.
- [2] Jan Borchers. *A Pattern Approach to Interaction Design*. John Wiley & Sons, Inc., New York, NY, USA, 2001.
- [3] T.H. Bui. Multimodal Dialogue Management - State of the art. Technical Report TR-CTI, Enschede, January 2006.
- [4] James O. Coplien. *A generative development-process pattern language*, pages 183–237. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.
- [5] Charles Dvorak, Judith Kiss, and Hiroshi Ota. Parameters describing the interaction with spoken dialogue systems. *ITU-T Recommendations*, pages 1–26, October 2005.
- [6] Tobias Heinroth and Dan Denich. Spoken Interaction within the Computed World: Evaluation of a Multitasking Adaptive Spoken Dialogue System. In *35th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2011)*. IEEE, 2011.

- [7] J Nielsen. Classification of dialog techniques. *ACM SIGCHI Bulletin*, 1987.
- [8] Erik G. Nilsson. Design patterns for user interface for mobile applications. *Advances in Engineering Software*, 40(12):1318–1328, December 2009.
- [9] Raquel O. Prates, Clarisse S. de Souza, and Simone D. J. Barbosa. Methods and tools: a method for evaluating the communicability of user interfaces. *interactions*, 7:31–38, January 2000.
- [10] Gustavo Rossi, Daniel Schwabe, and Fernando Lyardet. User interface patterns for hypermedia applications. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, AVI '00, pages 136–142. ACM, 2000.
- [11] A Rudnicky. An agenda-based dialog management architecture for spoken language systems. *IEEE Automatic Speech Recognition and ...*, 1999.
- [12] Dirk Schnelle. *Context Aware Voice User Interfaces for Workflow Support*. PhD thesis, Technische Universität Darmstadt, 2008.
- [13] Dirk Schnelle and Fernando Lyardet. Voice User Interface Design Patterns. In *EuroPLoP 2006 Conference Proceedings*, 2006.
- [14] Dirk Schnelle, Fernando Lyardet, and Tao Wei. Audio navigation patterns. In Uwe Zdun Andy Longshaw, editor, *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, pages 237–260. UVK Universitätsverlag Konstanz, 2005.
- [15] Dirk Schnelle-Walka. A Pattern Language for Error Management in Voice User Interfaces. In *EuroPLoP '10: Proceedings of the European Conference on Pattern Languages of Programs*, page 24, 2010.
- [16] Dirk Schnelle-Walka. I Tell You Something. In *EuroPLoP '11: Proceedings of the European Conference on Pattern Languages of Programs*, 2011.
- [17] Alksandra Tešanović. What is a pattern. In *Dr.ing. course DT8100 (prev. 78901 / 45942 / DIF8901) Object-oriented Systems*. IDA Department of Computer and Information Science, Linköping, Sweden, 2005.
- [18] Daniel Traum. Conversational Agency: The Trains-93 Dialogue Manager. *Proceedings of the Twente Workshop on Language Technology 11: Dialogue Management in Natural Language Systems*, pages 1–11, July 1996.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland